

C和C++程序员面试秘笈

精选最常见的C/C++面试真题

董山海　编著

完全从实战出发的C/C++面试技巧

- 紧扣面试精髓，一册在手，工作不愁
- 提供最全的C/C++面试题分类，帮你找到好工作
- 涵盖经典的笔试题+上机题，帮你整理知识库
- 解析考题的要点和技术原理，帮你温故而知新



人民邮电出版社
POSTS & TELECOM PRESS



C和C++程序员面试秘笈

精选最常见的C/C++面试真题

董山海 编著



完全从实战出发的C/C++面试技巧

- 紧扣面试精髓，一册在手，工作不愁
- 提供最全的C/C++面试题分类，帮你找到好工作
- 涵盖经典的笔试题+上机题，帮你整理知识库
- 解析考题的要点和技术原理，帮你温故而知新

人民邮电出版社
POSTS & TELECOM PRESS

目　录

封面
扉页
前言
第1章 C/C++程序基础
面试题1 看代码写输出——一般赋值语句
面试题2 看代码写输出——C++域操作符
面试题3 看代码写输出——i++和++i的区别
面试题4 i++与++i哪个效率更高
面试题5 选择编程风格良好的条件比较语句
面试题6 看代码写结果——有符号变量与无符号变量的值的转换
面试题7 不使用任何中间变量如何将a、b的值进行交换
面试题8 C++与C有什么不同
面试题9 如何理解C++是面向对象化的，而C是面向过程化的
面试题10 标准头文件的结构
面试题11 #include <head.h>和#include "head.h" 有什么区别
面试题12 C++中main函数执行完后还执行其他语句吗
第2章 预处理、const、static与sizeof
面试题1 预处理的使用
面试题2 用#define实现宏并求最大值和最小值
面试题3 宏定义的使用
面试题4 看代码写输出——宏参数的连接
面试题5 用宏定义得到一个字的高位和低位字节
面试题6 用宏定义得到一个数组所含的元素个数

面试题7 找错——const的使用
面试题8 说明const与#define的特点及区别
面试题9 C++中const有什么作用（至少说出3个）
面试题10 static有什么作用（至少说出2个）
面试题11 static全局变量与普通的全局变量有什么区别
面试题12 看代码写结果——C++类的静态成员
面试题13 使用sizeof计算普通变量所占空间大小
面试题14 使用sizeof计算类对象所占空间大小
面试题15 使用sizeof计算含有虚函数的类对象的空间大小
面试题16 使用sizeof计算虚拟继承的类对象的空间大小
面试题17 sizeof与strlen有哪些区别
面试题18 sizeof有哪些用途
面试题19 找错——使用strlen()函数代替sizeof计算字符串长度
面试题20 使用sizeof计算联合体的大小
面试题21 #pragma pack的作用
面试题22 为什么要引入内联函数
面试题23 为什么inline能很好地取代表达式形式的预定义
面试题24 说明内联函数使用的场合
面试题25 为什么不把所有的函数都定义成内联函数
面试题26 内联函数与宏有什么区别
第3章 引用和指针
面试题1 一般变量引用
面试题2 指针变量引用
面试题3 看代码找错误——变量引用
面试题4 如何交换两个字符串
面试题5 程序查错——参数引用
面试题6 参数引用的常见错误

面试题7 指针和引用有什么区别
面试题8 为什么传引用比传指针安全
面试题9 复杂指针的声明
面试题10 看代码写结果——用指针赋值
面试题11 指针加减操作
面试题12 指针比较
面试题13 看代码找错误——内存访问违规
面试题14 指针的隐式转换
面试题15 指针常量与常量指针的区别
面试题16 指针的区别
面试题17 找错——常量指针和指针常量的作用
面试题18 this指针的正确叙述
面试题19 看代码写结果——this指针
面试题20 指针数组与数组指针的区别
面试题21 找错——指针数组和数组指针的使用
面试题22 函数指针与指针函数的区别
面试题23 数组指针与函数指针的定义
面试题24 各种指针的定义
面试题25 代码改错——函数指针的使用
面试题26 看代码写结果——函数指针的使用
面试题27 typedef用于函数指针定义
面试题28 什么是“野指针”
面试题29 看代码查错——“野指针”的危害
面试题30 有了malloc/free，为什么还要new/delete
面试题31 程序改错——指针的初始化
面试题32 各种内存分配和释放的函数的联系和区别
面试题33 程序找错——动态内存的传递

面试题34 动态内存的传递
面试题35 比较分析两个代码段的输出——动态内存的传递
面试题36 程序查错——“野指针”用于变量值的互换
面试题37 内存的分配方式有几种
面试题38 什么是句柄
面试题39 指针与句柄有什么区别
第4章 字符串
面试题1 使用库函数将数字转换为字符串
面试题2 不使用库函数将整数转换为字符串
面试题3 使用库函数将字符串转换为数字
面试题4 不使用库函数将字符串转换为数字
面试题5 编程实现strcpy函数
面试题6 编程实现memcpy函数
面试题7 strcpy与memcpy的区别
面试题8 改错——数组越界
面试题9 分析程序——数组越界
面试题10 分析程序——打印操作可能产生数组越界
面试题11 编程实现计算字符串的长度
面试题12 编程实现字符串中子串的查找
面试题13 编程实现字符串中各单词的翻转
面试题14 编程判断字符串是否为回文
面试题15 编程实现stcmp库函数
面试题16 编程查找两个字符串的最大公共子串
面试题17 不使用printf，将十进制数以二进制和十六进制的形式输出
面试题18 编程实现转换字符串、插入字符的个数
面试题19 字符串编码例题

面试题20 反转字符串，但其指定的子串不反转
面试题21 编写字符串反转函数strrev
面试题22 编程实现任意长度的两个正整数相加
面试题23 编程实现字符串的循环右移
面试题24 删除指定长度的字符
面试题25 字符串的排序及交换
面试题26 编程实现删除字符串中所有指定的字符
面试题27 分析代码——使用strcat连接字符串
面试题28 编程实现库函数strcat
面试题29 编程计算含有汉字的字符串长度
面试题30 找出01字符串中0和1连续出现的最大次数
面试题31 编程实现字符串的替换
第5章 位运算与嵌入式编程
面试题1 位制转换
面试题2 看代码写出结果——位运算
面试题3 设置或清除特定的位
面试题4 计算一个字节里有多少bit被置1
面试题5 位运算改错
面试题6 运用位运算交换a、b两数
面试题7 列举并解释C++中的4种运算符转化以及它们的不同点
面试题8 用#define声明一个常数
面试题9 如何用C语言编写死循环
面试题10 如何访问特定位置的内存
面试题11 对中断服务代码的评论
面试题12 看代码写结果——整数的自动转换
面试题13 关键字static的作用是什么
面试题14 关键字volatile有什么含义

面试题15 判断处理器是Big_endian还是Little_endian
面试题16 评价代码片断——处理器字长
第6章 C++面向对象
面试题1 描述面向对象技术的基本概念
面试题2 判断题——类的基本概念
面试题3 选择题——C++与C语言相比的改进
面试题4 class和struct有什么区别
面试题5 改错——C++类对象的声明
面试题6 看代码写结果——C++类成员的访问
面试题7 找错——类成员的初始化
面试题8 看代码写结果——静态成员变量的使用
面试题9 与全局对象相比，使用静态数据成员有什么优势
面试题10 有哪几种情况只能用intialization list， 而不能用assignment
面试题11 静态成员的错误使用
面试题12 对静态数据成员的正确描述
面试题13 main函数执行前还会执行什么代码?
面试题14 C++中的空类默认会产生哪些类成员函数
面试题15 构造函数和析构函数是否可以被重载
面试题16 关于重载构造函数的调用
面试题17 构造函数的使用
面试题18 构造函数explicit与普通构造函数的区别
面试题19 explicit构造函数的作用
面试题20 C++中虚析构函数的作用是什么
面试题21 看代码写结果——析构函数的执行顺序
面试题22 复制构造函数是什么？什么是深复制和浅复制
面试题23 编译器与默认的copy constructor

面试题24 写一个继承类的复制函数
面试题25 复制构造函数与赋值函数有什么区别
面试题26 编写类String的构造函数、析构函数和赋值函数
面试题27 了解C++类各成员函数的关系
面试题28 C++类的临时对象
面试题29 复制构造函数和析构函数
面试题30 看代码写结果——C++静态成员和临时对象
面试题31 什么是临时对象？临时对象在什么情况下产生
面试题32 为什么C语言不支持函数重载而C++能支持
面试题33 判断题——函数重载的正确声明
面试题34 重载和覆写有什么区别
面试题35 编程题——MyString类的编写
面试题36 编程题——各类运算符重载函数的编写
面试题37 看代码写输出——new操作符重载的使用
第7章 C++继承和多态
面试题1 C++类继承的三种关系
面试题2 C++继承关系
面试题3 看代码找错——C++继承
面试题4 私有继承有什么作用
面试题5 私有继承和组合有什么相同点和不同点
面试题6 什么是多态
面试题7 虚函数是怎么实现的
面试题8 构造函数调用虚函数
面试题9 看代码写结果——虚函数的作用
面试题10 看代码写结果——虚函数
面试题11 虚函数相关的选择题
面试题12 为什么需要多重继承？它的优缺点是什么

面试题13 多重继承中的二义性
面试题14 多重继承二义性的消除
面试题15 多重继承和虚拟继承
面试题16 为什么要引入抽象基类和纯虚函数
面试题17 虚函数与纯虚函数有什么区别？
面试题18 程序找错——抽象类不能实例化
面试题19 应用题——用面向对象的方法进行设计
面试题20 什么是COM
面试题21 COM组件有什么特点
面试题22 如何理解COM对象和接口？
面试题23 简述COM、ActiveX和DCOM
面试题24 什么是DLL HELL
第8章 数据结构
面试题1 编程实现一个单链表的建立
面试题2 编程实现一个单链表的测长
面试题3 编程实现一个单链表的打印
面试题4 编程实现一个单链表节点的查找
面试题5 编程实现一个单链表节点的插入
面试题6 编程实现一个单链表节点的删除
面试题7 实现一个单链表的逆置
面试题8 寻找单链表的中间元素
面试题9 单链表的正向排序
面试题10 判断链表是否存在环型链表问题
面试题11 有序单链表的合并
面试题12 约瑟夫问题的解答
面试题13 编程实现一个双向链表的建立
面试题14 编程实现一个双向链表的测长

面试题15 编程实现一个双向链表的打印
面试题16 编程实现一个双向链表节点的查找
面试题17 编程实现一个双向链表节点的插入
面试题18 编程实现一个双向链表节点的删除
面试题19 实现有序双向循环链表的插入操作
面试题20 删除两个双向循环链表的相同结点
面试题21 编程实现队列的入队、出队、测长、打印
面试题22 队列和栈有什么区别
面试题23 简答题——队列和栈的使用
面试题24 选择题——队列和栈的区别
面试题25 使用队列实现栈
面试题26 选择题—栈的使用
面试题27 用C++实现一个二叉排序树
面试题28 使用递归与非递归方法实现中序遍历
面试题29 使用递归与非递归方法实现先序遍历
面试题30 使用递归与非递归方法实现后序遍历
面试题31 编写层次遍历二叉树的算法
面试题32 编写判别给定二叉树是否为二叉排序树的算法
第9章 排序
面试题1 编程实现直接插入排序
面试题2 编程实现希尔（Shell）排序
面试题3 编程实现冒泡排序
面试题4 编程实现快速排序
面试题5 编程实现选择排序
面试题6 编程实现堆排序
面试题7 实现归并排序的算法（使用自顶向下的方法）
面试题8 使用基数排序对整数进行排序

面试题9 选择题——各排序算法速度的性能比较
面试题10 各排序算法的时间复杂度的比较
第10章 泛型编程
面试题1 举例说明什么是泛型编程
面试题2 函数模板与类模板分别是什么
面试题3 使用模板有什么缺点？如何避免
面试题4 选择题——类模板的实例化
面试题5 解释什么是模板的特化
面试题6 部分模板特例化和全部模板特例化有什么区别
面试题7 使用函数模板对普通函数进行泛型化
面试题8 使用类模板对类进行泛型化
面试题9 通过类模板设计符合要求的公共类
第11章 STL（标准模板库）
面试题1 什么是STL
面试题2 具体说明STL如何实现vector
面试题3 看代码回答问题——vector容器中iterator的使用
面试题4 看代码找错——vector容器的使用
面试题5 把一个文件中的整数排序后输出到另一个文件中
面试题6 list和vector有什么区别
面试题7 分析代码问题并修正——list和vector容器的使用
面试题8 stl::deque是一种什么数据类型
面试题9 在做应用时如何选择vector和deque
面试题10 看代码找错——适配器stack和queue的使用
面试题11 举例说明set的用法
面试题12 举例说明map的用法
面试题13 STL中map内部是怎么实现的
面试题14 map和hashmap有什么区别

面试题15 什么是STL算法
面试题16 分析代码功能——STL算法的使用
面试题17 vector中的erase方法与algorithm中的remove有什么区别
面试题18 什么是auto_ptr（STL智能指针）？如何使用
面试题19 看代码找错——智能指针auto_ptr的使用
面试题20 智能指针如何实现
面试题21 使用std::auto_ptr有什么方面的限制
面试题22 如何理解函数对象
面试题23 如何使用bind1st和bind2nd
面试题24 实现bind1st的函数配接器
第12章 智力测试题
面试题1 元帅领兵
面试题2 两龟赛跑
面试题3 电视机的价格
面试题4 这块石头究竟有多重
面试题5 四兄弟的年龄
面试题6 爬楼梯
面试题7 3只砝码称东西
面试题8 称米
面试题9 比萨饼交易
面试题10 伊沙贝拉时装精品屋
面试题11 烧绳子的时间计算问题
面试题12 给工人的金条
面试题13 被污染的药丸
面试题14 称量罐头
面试题15 有20元钱可以喝到几瓶汽水

面试题16 判断鸟的飞行距离
面试题17 按劳取酬
面试题18 空姐分配物品
面试题19 消失的1元钱
面试题20 分物品
面试题21 称出4升的水
面试题22 通向诚实国和说谎国的路
面试题23 排序问题
面试题24 两个同一颜色的果冻
面试题25 怎样称才能用3次就找到球
面试题26 计算生日是哪一天
面试题27 3个女儿的年龄
面试题28 取回黑袜和白袜
面试题29 谁先击完40下鼠标
面试题30 聪明人是怎样发财的
面试题31 谁打碎了花瓶
面试题32 大有作为
面试题33 宴会桌旁
面试题34 过桥问题
面试题35 一句不可信的话
面试题36 海盗分宝石
面试题37 如何推算有几条病狗
面试题38 判断谁是盗窃犯
面试题39 向导
面试题40 扑克牌问题
面试题41 谁是机械师
面试题42 帽子的颜色

面试题43 两个大于1小于10的整数
面试题44 谁用1美元的纸币付了糖果钱
面试题45 究竟有哪些人参加了会议
面试题46 小虫
面试题47 相遇
面试题48 约会
面试题49 30秒答题
面试题50 1分钟答题
面试题51 现代斯芬克斯之谜
面试题52 所有开着的灯的编号
版权

C和C++程序员面试秘笈

董山海　编著

人民邮电出版社

北京

前言

本书的思路

从涉及的开发领域来说，C/C++无疑是目前所有语言之中的翘楚，在Windows编程、嵌入式编程、各种通信编程中都有 C/C++的影子。因为涉及的领域众多，社会对 C/C++的人才需求也越来越多。不管招聘的职位是嵌入式下的开发还是 Windows 下的开发，熟悉 C/C++语言基础是必需的招聘要求。

虽然全世界每个月都可能会出现新的语言，但从 TIOBE 世界编程语言排行榜的数据来看，从2009年到现在，C/C++一直都在前3甲中。不论历史的车轮如何滚滚向前，学好C/C++永远不会落后。

本书针对的是刚毕业或刚学完 C/C++的入门读者，目的是帮助读者找到更好的工作并复习所学的C/C++基础。

本书的特点

本书全面讲解了 C/C++面试的各种知识点，并对一些重点和难点进行了细致的分析。特点如下：

本书条理清晰，章节内容由易至难，由浅入深，先从C 程序设计入手，再详细讲解C++面向对象的高级特性，最后讲解泛型编程和STL。

本书对于每个面试例题都有详细的讲解以及源代码分析。

书中还讨论了数据结构和算法，给出了一些经典的数据结构和算法的 C 语言实现，便于读者快速掌握面试中所需的知识。

针对面试中出现越来越多的智力测试部分，本书对大部分常见的智力题和逻辑思维题进行了归类及分析解答。

本书的内容

本书内容突出了在 C/C++面试中或者是项目开发中，必须掌握的技能和容易忽略的内容。对 C/C++面试者来说，可以快速掌握面试过程中考查的知识点，减少面试准备时间，提高面试成功率。本书共分为12章，有310余道面试题。

第1章 C/C++程序基础

本章介绍了赋值语句、递增语句、类型转换、数据交换等程序设计的基本概念。希望读者在面试之前复习这些概念，并重视那些比较细微却基础的考点。

第2章 预处理、const、static 与sizeof

本章介绍了 C/C++设计语言中的难点，这也是各个企业面试中反复出现的考点。尤其是static和sizeof，它们在许多笔试以及面试的题目中都出现过。

第3章 引用和指针

本章介绍了引用和指针，这是 C/C++的基础，又是学习过程中最难翻越的一道坎。本章通过编写实例的方式讲解了数组指针、函数指针、常量指针、指针传值、多维指针等容易让读者混淆的概念。

第4章 字符串

本章介绍了字符串的应用和字符串处理的一些函数。字符串是笔试以及面试的热门考点，通过字符串测试可以考察程序员的编程规范以及编程习惯。其中也包括了许多知识点，例如内存越界、指针与数组操作等。许多公司在面试时会要求应试者写一段 strcpy 复制字符串或字符串子串操作的程序。

第5章 位运算与嵌入式编程

本章介绍了在位运算与嵌入式开发中容易出现的面试题。C 语言是嵌入式开发所必需的编程技术，因此在招聘嵌入式系统程序员时会要求必须非常熟悉C/C++语言。

第6章 C++面向对象

C语言是面向过程的，而C++作为C语言的超集支持，它是面向对象的。面向对象（ObjectOriented）是当前计算机界关心的重点，它是当今软件开发方法的主流，因此也是各大公司的重要考点。

第7章 C++继承和多态

继承和多态是 C++面向对象程序设计的关键。继承机制使得派生类能够获得基类的成员数据和方法，只需要在派生类中增加基类没有的成员。多态是建立在继承的基础上的，它使用了C++编译器最核心的一个技术，即动态绑定技术。这些都是面试必考题型。

第8章 数据结构

算法的设计依赖于数据的逻辑结构，算法的实现依赖于数据的存储结构，所以数据结构选择得好坏，对程序的质量影响甚大。掌握基本的数据结构知识，是程序设计水平提高的必要条件。算法和数据结构也是面试中的必考题型。

第9章 排序

排序法属于算法中解决数据排列问题的解决方案。本章演示了插入排序、选择排序、交换排序、归并排序和分配排序的实现过程。每一种排序法都可能成为一道面试题。

第10章 泛型编程

泛型编程是一种新的编程思想，它基于模板技术，有效地将算法与数据结构分离，降低了模块间的耦合度。本章演示了泛型在 C/C++中的应用，如函数模板和类模板。这些内容是难点，也是考点。

第11章 STL

STL 是标准模板库，它涵盖了常用的数据结构和算法，并且具有跨平台的特点。将泛型编程思想和STL库用于系统设计中，明显降低了开发强度，提高了程序的可维护性及代码的可重用性。这也是越来越多的笔试和面试中考查STL相关知识的原因。

第12章 智力测试题

有很多有趣的逻辑思考题目出现于跨国企业的招聘面试中，它对考查一个人的思维方式及思维方式的转变能力有极其明显的作用。据一些研究显示，这样的能力往往也与工作中的应变与创新状态息息相关。本章面试题不一定都有固定的答案，有时候只是考查应聘者的逻辑思维。

本书的读者群

即将步入IT 行业的应届大学毕业生；

有一定工作经验但C/C++编程基础不好的程序员；

想跳槽又怕找不到适合自己的工作的C/C++程序员；

刚从培训机构学习完C/C++的入门者；

C/C++培训机构的课后阅读图书；

C/C++语言爱好者。

编者

第1章 C/C++程序基础

作为程序员，你在求职时，公司会询问你的项目经验，例如你做过什么类型的项目、担任的是何种角色，以及做项目时如何与他人沟通，等等。除此之外，当然还要考查你的编程能力。这里包括你的编程风格，以及你对于赋值语句、递增语句、类型转换、数据交换等程序设计基本概念的理解。因此，最好在考试之前复习这些程序设计的基本概念，并且要特别重视那些比较细致的考点问题。本章列出了一些涉及C/C++程序设计基本概念的考题，希望读者在读完后能有所收获。

面试题1 看代码写输出——一般赋值语句

考点：一般赋值语句的概念和方法

出现频率：★★★

```
1    #include <stdio.h>
2
3    int main(void)
4    {
5        int x = 3, y, z;
6
7        x *= (y = z = 4); printf("x = %d\n", x);
8
9        z = 2;
10         x = (y = z); printf("x = %d\n", x);
11         x = (y == z); printf("x = %d\n", x);
12         x = (y & z); printf("x = %d\n", x);
13         x = (y && z); printf("x = %d\n", x);
14
15         y = 4;
16         x = (y | z); printf("x = %d\n", x);
17         x = (y || z); printf("x = %d\n", x);
18
19         x = (y == z)? 4: 5;
20         printf("x = %d\n", x);
21
```

```
22        x = (y == z)? 1: (y < z)? 2: 3;
23        printf("x = %d\n", x);
24
25        return 0;
26    }
```

【解析】

程序的说明如下：

程序执行至第8 行时，x 的值为3，y 和z 未被初始化。此行的执行顺序是首先执行z=4，然后执行y=z，最后执行x*=y。因此x的值为3*4=12。

程序执行至第 10 行时，z 的值为 2。此行的执行顺序是首先执行y=z，然后执行x=y。因此x的值为2。

程序执行至第11 行时，y 和z 的值都为2。此行的执行顺序是首先执行y==z，比较y和z的值是否相等，然后将比较的结果赋给x。因此x的值为1。

程序执行至第12 行时，y和z 的值都为2。此行把y和z 做按位与（&）运算的结果赋给变量x。y 和z 的二进制都是10，因此y & z 的结果为二进制10。因此x的值为2。

程序执行至第13 行时，y和z 的值都为2。此行把y和z 做逻辑与（&&）运算的结果赋给变量x。此时y和z 的值都不是0，因此y && z 的结果为1。因此x 的值为1。

程序执行至第16 行时，y的值为4，z 的值为2。此行把y和z 做按位或（|）运算的结果赋给变量x。此时y和z的二进制表示分别为100和010，因此y|z的结果为110。因此x的值为110，十进制表示为6。

程序执行至第17 行时，y 的值为4，z 的值为2。此行把y 和z 做逻辑或（||）运算的结果赋给变量x。此时y和z的值都不是0，因此y||z的结果为1。因此x的值为1。

程序执行至第19 行时，y的值为4，z 的值为2。此行首先比较y 和z的大小是否相等，如果相等，则将x取4和5的前者，否则x取4和5的后者。在这里，y不等于z，因此x的值为5。

程序执行至第22 行时，y的值为4，z 的值为2。此行首先比较y 和z大小是否相等，如果相等，x取1，否则，判断y是否大于z，如果是，则取2，否则取3。在这里，y的值大于z的值，因此x的值为3。

总结：这个考题只是考查各种基本的赋值运算。这里，读者要注意位运算与逻辑运算的区别，以及三元操作符的用法。通过程序代码 17 行以及 19 行的举例，我们可以发现三元操作符有时可以代替条件判断if/else/else if的组合。

【答案】

x = 12

x = 2

x = 1

x = 2

x = 1

x = 6

x = 1

x = 5

x = 3

面试题2 看代码写输出——C++域操作符

考点：C++域操作符的使用

出现频率：★★★

请指出下面这个程序在C和C++中的输出分别是什么。

```
1    #include <stdio.h>
2
3    int value = 0;
4
5    void printvalue()
6    {
7        printf("value = %d\n", value);
8    };
9
10   int main()
11   {
12       int value = 0;
13
14       value = 1;
15       printf("value = %d\n", value);
16
17       ::value = 2;
18       printvalue();
19
20       return 0;
```

21　}

【解析】

如果将文件保存为后缀名为.c的文件，则在Visual C++ 6.0中不能通过编译并且提示17行有语法错误。而如果文件保存为后缀名为.cpp的文件，则在Visual C++ 6.0中就能顺利通过编译并且运行。

这段程序有两个变量，其名字都是value。不同的是，其中一个是在main函数之前就声明的全局变量，而另外一个是在main函数内部声明的局部变量。这两个变量的作用域是不一样的。

这里要注意：在函数printvalue里打印的是全局变量的值，在main函数的15行打印的是局部变量的值。这是因为在main函数里的局部变量value引用优先。在C++中可以通过域操作符“::”来直接操作全局变量（代码的17行操作的value是全局变量），但是在C中不支持这个操作符，因此会报错。注意：在 C 中不推荐这种局部变量与全局变量同名的设计方式。

【答案】

在C中编译不能通过，并指示17行符号错误。

在C++中的输出如下：

value=1（局部变量value）

value=2（全局变量value）

面试题3 看代码写输出——i++和++i的区别

考点：i++和++i的区别

出现频率：★★★★★

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       int i=8;
6
7       printf("%d\n",++i);
8       printf("%d\n",--i);
9       printf("%d\n",i++);
10      printf("%d\n",i--);
11      printf("%d\n",-i++);
12      printf("%d\n",-i--);
13      printf("------\n");
14
15      return 0;
16  }
```

【解析】

程序的说明如下：

程序第7 行，此时i 的值为8。这里先i 自增1，再打印i 的值。因此输出9，并且i的值也变为9。

程序第8 行，此时i 的值为9。这里先i 自减1，再打印i 的值。因此

输出8，并且i的值也变为8。

程序第9 行，此时i 的值为8。这里先打印i 的值，再i 自增1。因此输出8，并且i的值也变为9。

程序第10 行，此时i 的值为9。这里先打印i 的值，再i 自减1。因此输出9，并且i的值也变为8。

程序第11 行，此时i 的值为8。这里的“-”表示负号运算符。因此先打印-i 的值，再i自增1。因此输出-8，并且i的值也变为9。

程序第12 行，此时i 的值为9。这里的第一个“-”表示负号运算符，后面连在一起的两个“-”表示自减运算符。因此先打印-i的值，再i自减1。因此输出-9，并且i的值也变为8。

【答案】

9

8

8

9

-8

-9

面试题4 i++与++i哪个效率更高

考点：i++和++i的效率比较

出现频率：★★★

【解析】

在这里声明，简单地比较前缀自增运算符和后缀自增运算符的效率是片面的，因为存在很多因素影响这个问题的答案。首先考虑内建数据类型的情况：如果自增运算表达式的结果没有被使用，而是仅仅简单地用于增加一员操作数，答案是明确的，前缀法和后缀法没有任何区别，编译器的处理都应该是相同的，很难想象得出有什么编译器实现可以别出心裁地在二者之间制造任何差异。我们看看下面这个程序。

```
1    #include <stdio.h>
2
3    int main()
4    {
5        int i = 0;
6        int x = 0;
7
8        i++;
9        ++i;
10        x = i++;
11        x = ++i;
12
13        return 0;
14    }
```

上面的代码在VC++ 6.0 中编译，得到的汇编如下。

```
; Line 5
   mov DWORD PTR _i$[ebp], 0
; Line 6
   mov DWORD PTR _x$[ebp], 0
; Line 8
   mov eax, DWORD PTR _i$[ebp]
   add eax, 1
   mov DWORD PTR _i$[ebp], eax
; Line 9
   mov ecx, DWORD PTR _i$[ebp]
   add ecx, 1
   mov DWORD PTR _i$[ebp], ecx
; Line 10
   mov    edx, DWORD PTR _i$[ebp]
   mov    DWORD PTR _x$[ebp], edx
   mov    eax, DWORD PTR _i$[ebp]
   add    eax, 1
   mov    DWORD PTR _i$[ebp], eax
; Line 11
   mov    ecx, DWORD PTR _i$[ebp]
   add    ecx, 1
   mov    DWORD PTR _i$[ebp], ecx
   mov    edx, DWORD PTR _i$[ebp]
   mov    DWORD PTR _x$[ebp], edx
```

代码段第8行和第9 行生成的汇编代码分别对应Line 8 和Line 9 下对应的汇编代码，可以看到3个步骤几乎完全一样。

代码段第10～11 行生成的汇编代码分别对应Line 10 和Line 11 下对应的汇编代码，可以看到都是5个步骤，只是在加1的先后顺序上有一些区别，效率也是完全一样的。

由此说明，考虑内建数据类型时，它们的效率差别不大（去除编译器优化的影响）。所以在这种情况下，我们大可不必关心。

现在让我们再考虑自定义数据类型（主要是指类）的情况。此时我们不需要再做很多汇编代码的分析了，因为前缀式（++i）可以返回对象的引用，而后缀式（i++）必须返回对象的值，所以导致在大对象的时候产生了较大的复制开销，引起效率降低。因此处理使用者自定义类型（注意不是指内建类型）的时候，应该尽可能地使用前缀式递增/递减，因为它天生“体质”较佳。

【答案】

内建数据类型的情况，效率没有区别。

自定义数据类型的情况，++i效率较高。

面试题5 选择编程风格良好的条件比较语句

考点：良好的编程风格

出现频率：★★★★

A．假设布尔变量名字为flag，它与零值比较的标准if语句如下。

第一种：

```
1    if (flag == TRUE)
2    if (flag == FALSE)
```

第二种：

```
1    if (flag)
2    if (!flag)
```

B．假设整型变量的名字为value，它与零值比较的标准if语句如下。

第一种：

```
1    if (value == 0)
2    if (value != 0)
```

第二种：

```
1    if (value)
2    if (!value)
```

C．假设浮点变量的名字为 x，它与 0.0的比较如下。

第一种：

```
1    if (x == 0.0)
2    if (x != 0.0)
```

第二种：

```
1    if ((x >= -EPSINON) && (X <= EPSINON))
```

2　if ((x < -EPSINON) || (X > EPSINON))

其中，EPSINON是允许的误差（精度）。

D．指针变量 p与 0的比较如下。

第一种：

1　if (p == NULL)

2　if (p != NULL)

第二种：

1　if (p == 0)

2　if (p != 0)

【解析】

A 的第二种风格较良好。根据布尔类型的语义，零值为“假”（记为FALSE），任何非零值都是“真”（记为TRUE）。TRUE的值究竟是什么并没有统一的标准。例如Visual C++将TRUE定义为1，而Visual Basic 则将TRUE 定义为-1。因此不可将布尔变量直接与TRUE、FALSE进行比较。

B 的第一种风格较良好，第二种风格会让人误解value 是布尔变量，应该将整型变量用“==”或“！=”直接与0比较。

C 的第二种风格较良好。注意：无论是float 还是double 类型的变量，都有精度限制。所以一定要避免将浮点变量用“==”或“！=”与数字比较，应该设法转化成“>=”或“<=”形式。

D 的第一种风格较良好，指针变量的零值是“空”（记为 NULL）。尽管 NULL的值与0相同，但是两者意义不同。用p与NULL显式比较，强调p是指针变量。

如用p与0比较，容易让人误解p是整型变量。

面试题6 看代码写结果——有符号变量与无符号变量的值的转换

考点：有符号变量与无符号变量的区别和联系

出现频率：★★★★

```
1    #include <stdio.h>
2
3    char getChar(int x, int y)
4    {
5        char c;
6        unsigned int a = x;
7
8        (a + y > 10)? (c = 1): (c = 2);
9        return c;
10   }
11
12   int main(void)
13   {
14       char c1 = getChar(7, 4);
15       char c2 = getChar(7, 3);
16       char c3 = getChar(7, -7);
17       char c4 = getChar(7, -8);
18
19       printf("c1 = %d\n", c1);
20       printf("c2 = %d\n", c2);
```

```
21        printf("c3 = %d\n", c3);
22        printf("c4 = %d\n", c4);
23
24        return 0;
25    }
```

【解析】

首先说明getChar()函数的作用：它有两个输入参数，分别是整型的x和y。在函数体内，把参数x的值转换为无符号整型后再与y相加，其结果与10进行比较，如果大于10，则函数返回1，否则返回2。在这里，我们要注意：当表达式中存在有符号类型和无符号类型时，所有的操作数都自动转换成无符号类型。因此，这里由于a是无符号数，在代码第8行中， y值会首先自动转换成无符号的整数，然后与a相加，最后再与10进行比较。以下是在main函数中各调用getChar()函数的分析。

代码第14 行，传入的参数分别为7 和4，两个数相加后为11，因此c1 返回1。

代码第15 行，传入的参数分别为7 和3，两个数相加后为10，因此c2 返回2。

代码第16 行，传入的参数分别为7 和-7，-7 首先被转换成一个很大的数，然后与7相加后正好溢出，其值为0，因此c3返回2。

代码第17 行，传入的参数分别为7 和-8，-8 首先被转换成一个很大的数，然后与7相加。两个数相加后为很大的整数（差1就正好溢出了），因此c4返回1。

我们可以看到，由于无符号整数的特性，getChar()当参数x为7时，如果y等于区间[-7,3]中的任何整数值，getChar()函数都将返回2。当y的值在区间[-7,3]之外时，函数返回-1。

总之，我们在看表达式时要很小心地注意符号变量与无符号变量之间的转换、占用不同字节内存的变量之间的赋值等操作，否则可能会出

现我们意想不到的结果。

【答案】

c1 = 1

c2 = 2

c3 = 2

c4 = 1

面试题7 不使用任何中间变量如何将a、b的值进行交换

考点：两个变量的值的交换方法

出现频率：★★★★

【解析】

请参考以下C++程序代码。

```
1   #include <stdio.h>
2
3   void swap1(int& a, int& b)
4   {
5       int temp = a;    //使用局部变量temp完成交换
6       a = b;
7       b = temp;
8   };
9
10  void swap2(int& a, int& b)
11  {
12      a=a+b;    //使用加减运算完成交换
13      b=a-b;
14      a=a-b;
15  };
16
17  void swap3(int& a, int& b)
18  {
```

```
19      a^=b;    //使用异或运算完成交换
20      b^=a;
21      a^=b;
22  };
23
24  int main(void)
25  {
26      int a1 = 1, b1 = 2;
27      int a2 = 3, b2 = 4;
28      int a3 = 5, b3 = 6;
29      int a = 2147483647, b = 1;
30
31      swap1(a1, b1);    //测试使用临时变量进行交换的版本
32      swap2(a2, b2);    //测试使用加减运算进行交换的版本
33      swap3(a3, b3);    //测试使用异或运算进行交换的版本
34
35      printf("after swap...\n");
36      printf("a1 = %d, b1 = %d\n", a1, b1);
37      printf("a2 = %d, b2 = %d\n", a2, b2);
38      printf("a3 = %d, b3 = %d\n", a3, b3);
39
40      swap2(a, b);
41      printf("a = %d, b = %d\n", a, b);
42
43      return 0;
44  }
```

以上的C++程序中有3个swap函数，都是采用引用传参的方式。

swap1()采用的是我们在许多教科书里看到的方式，用一个局部变量temp 保存其中一个值来达到交换目的。当然，这种方式不是本题要求的答案。

swap2()采用的是一种简单的加减算法来达到交换a、b 的目的。这种方式的缺点是做a+b和a-b运算时可能会导致数据溢出。

swap3()采用了按位异或的方式交换a、b。按位异或运算符“^”的功能是将参与运算的两数各对应的二进制位相异或，如果对应的二进制位相同，则结果为0，否则结果为1。这样运算3次即可交换a、b的值。

代码第31～32行做了调用3种swap函数的举例，注意第40行的调用，这里在swap2函数栈中的运算会有数据溢出发生。我们知道，在32位平台下，int占4个字节内存，其范围是-2147483648～2147483647，因此2147483647加1就变成了-2147483648。不过通过运行结果我们可以看到，虽然产生了溢出，但是交换操作依然是成功的。下面是程序运行结果。

after swap…

a1 = 2, b1 = 1

a1 = 4, b1 = 3

a1 = 6, b1 = 5

a1 = 1, b1 = 2147483647

【答案】

采用程序代码中swap2和swap3的交换方式。swap2有可能发生数据溢出的缺点。相比于swap2，推荐swap3，采用按位异或的方式。

面试题8 C++与C有什么不同

考点：C和C++的联系与区别

出现频率：★★★★

【答案】

C是一个结构化语言，它的重点在于算法和数据结构。对语言本身而言，C是C++的子集。C程序的设计首要考虑的是如何通过一个过程，对输入进行运算处理，得到输出。对于C++，首要考虑的是如何构造一个对象模型，让这个模型能够配合对应的问题，这样就可以通过获取对象的状态信息得到输出或实现过程控制。

因此，C与C++的最大区别在于，它们用于解决问题的思想方法不一样。

C实现了C++中过程化控制及其他相关功能。而在C++中的C，相对于原来的C还有所加强，引入了重载、内联函数、异常处理等。C++更是拓展了面向对象设计的内容，如类、继承、虚函数、模板和包容器类等。

在 C++中，不仅需要考虑数据封装，还需要考虑对象粒度的选择、对象接口的设计和继承、组合与继承的使用等问题。

相对于C，C++包含了更丰富的设计概念。

面试题9 如何理解C++是面向对象化的，而C是面向过程化的

考点：C++与C的区别

出现频率：★★★

【答案】

C 是面向过程化的，但是 C++不是完全面向对象化的。在 C++中也完全可以写出与 C一样过程化的程序，所以只能说C++拥有面向对象的特性。Java是真正面向对象化的。

面试题10 标准头文件的结构

为什么标准头文件都有类似以下的结构？

考点：标准头文件中一些通用结构的理解

出现频率：★★★★

```
1    #ifndef __INCvxWorksh
2    #define __INCvxWorksh
3    #ifdef __cplusplus
4    extern "C" {
5    #endif
6    /*...*/
7    #ifdef __cplusplus
8    }
9    #endif
10   #endif /* __INCvxWorksh */
```

【解析】

显而易见，代码第1、2、10行的作用是防止该头文件被重复引用。代码第3行的作用是表示当前使用的是C++编译器。如果要表示当前使用的是C编译器，可以这样指定：

```
1    #ifdef __STDC__
```

那么代码第4～8 行中的extern "C"有什么作用呢？

extern "C"包含双重含义。

首先，被它修饰的目标是“extern”的。也就是告诉编译器，其声明的函数和变量可以在本模块或其他模块中使用。通常，在模块的头文件中对本模块提供给其他模块引用的函数和全局变量以关键字extern声

明。例如，当模块B欲引用该模块A中定义的全局变量和函数时，只需包含模块A的头文件即可。这样，模块B中调用模块A中的函数时，在编译阶段，模块B虽然找不到该函数，但是并不会报错；它会在连接阶段中从模块A编译生成的目标代码中找到此函数。

其次，被它修饰的目标是“C”的，意思是其修饰的变量和函数是按照C语言方式编译和连接的。我们来看看C++中对类似C的函数是怎样编译的。作为一种面向对象的语言， C++支持函数重载，而过程式语言 C则不支持。函数被 C++编译后在符号库中的名字与 C语言的不同。例如下面两个函数：

1　void foo(int x, int y)；

2　void foo(int x, float y)；

这两个函数编译生成的符号是不相同的，前者可能为_foo_int_int 之类，而后者可能为_foo_int_float之类。可以发现，这样的名字包含了函数名、函数参数数量及类型信息，C++就是靠这种机制来实现函数重载的。这样，如果在C中连接C++编译的符号时，就会因找不到符号问题发生连接错误。

如果加extern "C"声明后，模块编译生成foo 的目标代码时，就不会对其名字进行特殊处理，采用了C语言的方式，也就是_foo之类，不会加上后面函数参数数量及类型信息相关的那一串了。因此extern "C"是C++编译器提供的与C 连接交换指定的符号，用来解决名字匹配问题。

【答案】

代码第1、2、10行的作用是防止该头文件被重复引用。

代码第3行的作用是表示当前使用的是C++编译器。

代码第4～8行中的extern "C"是C++编译器提供的与C 连接交换指定的符号，用来解决名字匹配问题。

面试题11 #include <head.h>和#include "head.h" 有什么区别

考点：头文件引用中<>与""的区别

出现频率：★★★★

【答案】

尖括号< >表明这个文件是一个工程或标准头文件。查找过程会首先检查预定义的目录，我们可以通过设置搜索路径环境变量或命令行选项来修改这些目录。

如果文件名用一对引号括起来，则表明该文件是用户提供的头文件，查找该文件时将从当前文件目录（或文件名指定的其他目录）中寻找文件，然后在标准位置寻找文件。

面试题12 C++中main函数执行完后还执行其他语句吗

考点：atexit()函数的使用

出现频率：★★★★

【解析】

很多时候，我们需要在程序退出的时候做一些诸如释放资源的操作，但程序退出的方式有很多种，例如main()函数运行结束，在程序的某个地方用exit()结束程序，用户通过Ctrl+C等操作发信号来终止程序，等等，因此需要有一种与程序退出方式无关的方法来进行程序退出时的必要处理。方法就是用atexit()函数来注册程序正常终止时要被调用的函数。

atexit()函数的参数是一个函数指针，函数指针指向一个没有参数也没有返回值的函数。atexit()的函数原型是：

```
1    int atexit (void (*)(void));
```

在一个程序中最多可以用atexit()注册32个处理函数，这些处理函数的调用顺序与其注册的顺序相反，即最先注册的最后调用，最后注册的最先调用。请看下面的程序代码。

```
1    #include<stdlib.h>                //使用atexit()函数必须包含头文件stdlib.h
2    #include<stdio.h>
3
4    void fn1(void);
5    void fn2(void);
6
```

```
7   int main(void)
8   {
9      atexit(fn1);                  //使用atexit注册fn1()函数
10       atexit(fn2);                  //使用atexit注册fn2()函数
11       printf("main exit...\n");
12       return 0;
13   }
14
15   void fn1()
16   {
17       printf("calling fn1()...\n");   //fn1()函数打印内容
18   }
19
20   void fn2()
21   {
22       printf("calling fn2()...\n");   //fn2()函数打印内容
23   }
```

上面的程序代码在main函数中调用atexit()函数依次注册了fn1()和fn2()函数。运行这个程序，我们可以得到下面的输出。

```
main exit…
calling fn1()…
calling fn2()…
```

在这里，fn2()与 fn1()在 main()函数结束后被依次调用，并且它们被调用的顺序与它们在main()函数被注册的顺序相反。

【答案】

可以用 atexit()函数来注册程序正常终止时要被调用的函数，并且在main()函数结束时，调用这些函数的顺序与注册它们的顺序相反。

第2章 预处理、const、static与sizeof

本章要讨论的这些问题都是 C/C++设计语言中的难点，并且是各个企业面试中反复出现的考点。尤其是static和sizeof，它们在许多笔试以及面试的题目中都出现过，因此本章将详细讨论这些问题。

面试题1 预处理的使用

看下面的代码并写出结果。

考点：#ifdef、#else、#endif在程序中的使用

出现频率：★★★

```
1    #include <stdio.h>
2    #include <stdlib.h>
3
4    #define DEBUG
5
6    int main()
7    {
8        int i = 0;
9        char c;
10
11       while(1) {
12           i++;
13           c = getchar();
14           if (c != '\n') {
15               getchar();
16           }
17           if (c == 'q' || c == 'Q'){
18   #ifdef DEBUG
19               printf("we got:%c, about to exit.\n", c);
20   #endif
```

```
21                break;
22            } else {
23                printf("i = %d", i);
24  #ifdef DEBUG
25                printf(", we got:%c", c);
26  #endif
27                printf("\n");
28            }
29        }
30
31        return 0;
32  }
```

【解析】

在代码第4行，首先定义了名为DEBUG的预处理器常量，而后分别在第18行和第24行用#ifdef判断DEBUG是否被定义了；如果被定义了，就进行printf输出信息。在这里传给main的代码如下。

```
6   int main()
7   {
8       int i = 0;
9       char c;
10
11        while(1) {
12            i++;
13            c = getchar();
14            if (c != '\n') {
15                getchar();
16            }
```

```
17          if (c == 'q' || c == 'Q'){
19                printf("we got:%c, about to exit.\n", c);
21                break;
22          } else {
23                printf("i = %d", i);
25                printf(", we got:%c", c);
27                printf("\n");
28          }
29       }
30
31       return 0;
32    }
```

根据上面展开之后的代码段，容易看出这个程序就是从终端每次读一个字符，如果字符为q或者Q，程序退出，否则打印收到的字符。执行结果如下所示。

输入：A

输出：i = 1, we got:A

输入：a

输出：i = 2, we got:a

输入：q

输出：we got:q, about to exit.

以上输出含有“we got”字符串的，均为#ifdef 与#endif之间的打印消息。

如果注释第4行，即DEBUG没有被定义，那么#ifdef与#endif之间的打印消息将不会被输出。

面试题2 用#define实现宏并求最大值和最小值

考点：#define宏定义的使用

出现频率：★★★★

【解析】

以下为实现代码。

```
1   #define   MAX( x, y ) ( ((x) > (y)) ? (x) : (y) )
2   #define   MIN( x, y ) ( ((x) < (y)) ? (x) : (y) )
```

在这里，MAX(x,y)表示x和y的最大值，MIN(x,y)表示x和y的最小值。此题有以下几个目的。

（1）#define在宏上应用的基本知识。

（2）三元运算符（? :）的知识。这个运算符能产生比 if-else 更优化的代码，并且书写上更加简洁明了。

（3）在宏中需要把参数小心地用括号括起来。因为宏只是简单的文本替换，如果不注意，很容易引起歧义。

面试题3 宏定义的使用

写出下面代码的输出结果。

考点：使用#define宏定义时需要注意的地方

出现频率：★★★★

```
1   #include <stdio.h>
2   #define SQR(x) (x*x)
3
4   int main()
5   {
6       int a, b = 3;
7       a = SQR(b+2);
8       printf("a = %d\n", a);
9       return 0;
10  }
```

【解析】

这里定义的SQR(x)显然是想要获得x的二次方，在第7行中调用的参数为b+2，原本想将a赋为(b+2)*(b+2)，也就是5的二次方，即25。但是由于宏定义展开是在预处理时期，也就是在编译之前，此时b并没有被赋值，这时的b只是一个符号。因此在第7行被展开成：

```
a = (b+2*b+2);
```

于是程序执行后，可以看到a被赋成11。

为了达到原来的目的，可以将SQR(x)改成如下定义。

```
#define SQR(x) ((x)*(x))
```

这样在第7行会被展开成：

a = ((b+2)*(b+2));

程序执行后，a被赋成25。

【答案】

a = 11

面试题4 看代码写输出——宏参数的连接

考点：如何连接宏参数

出现频率：★★★

```
1   #include <stdio.h>
2
3   #define STR(s)    #s
4   #define CONS(a,b)    (int)(a##e##b)
5
6   int main()
7   {
8       printf(STR(vck));
9       printf("\n");
10      printf("%d\n", CONS(2,3));
11
12      return 0;
13  }
```

【解析】

在这个程序中，我们使用#把宏参数变为一个字符串，用##把两个宏参数贴合在一起。

第3行中STR(s)定义的是一个参数s表示的字符串。在第8行的调用中，STR(vck)实际表示就是字符串"vck"。第4行中CONS(a,b)定义的是一个将参数a与b按aeb连接起来的一个整型值。在第 10 行的调用中，CONS(2,3)实际表示就是整型值 2e3，也就是十进制数2000。

【答案】

vck

2000

面试题5 用宏定义得到一个字的高位和低位字节

考点：宏定义与位运算的使用

出现频率：★★★★

【解析】

代码如下。

```
1  #define  WORD_LO(xxx)  ((byte) ((word)(xxx) & 255))
2  #define  WORD_HI(xxx)  ((byte) ((word)(xxx) >> 8))
```

一个字由两个字节组成。因此WORD_LO(xxx)取参数xxx的低8位，WORD_HI(xxx)取参数xxx的高8位。

【答案】

```
#define WORD_LO(xxx) ((byte) ((word)(xxx) & 255))
#define WORD_HI(xxx) ((byte) ((word)(xxx) >> 8))
```

面试题6 用宏定义得到一个数组所含的元素个数

考点：宏定义与sizeof的使用

出现频率：★★★★

【解析】

代码如下。

```
#define ARR_SIZE(a) (sizeof((a)) / sizeof((a[0])))
```

假设一个数组定义如下。

```
int array[100];
```

它含有100个int型的元素。如果int为4个字节，那么这个数组总共有400个字节。sizeof(array)为总大小，即400个字节；sizeof(array[0])为一个int大小，即4个字节。两个大小相除就是 100，也就是数组的元素个数。这里，为了保证宏定义不会发生“二义性”，在a以及a[0]上都加了括号。

【答案】

```
#define ARR_SIZE(a) (sizeof((a)) / sizeof((a[0])))
```

面试题7 找错——const的使用

考点：const使用时的注意点

出现频率：★★★★

```
1    #include <stdio.h>
2
3    int main()
4    {
5        const int x = 1;
6        int b = 10;
7        int c = 20;
8
9        const int* a1 = &b;
10       int* const a2 = &b;
11       const int* const a3 = &b;
12
13       x = 2;
14
15       a1 = &c;
16       *a1 = 1;
17
18       a2 = &c;
19       *a2 = 1;
20
21       a3 = &c;
```

```
22        *a3 = 1;
23
24        return 0;
25    }
```

【解析】

程序说明如下。

代码第13 行，由于变量x 为整型常量，因此不能改变x 的值。在这里会出现编译错误，并提示“l-value specifies const object”，也就是说，等号左边的是常量对象。如果在代码第5行没有给x初始化，那么x就是一个随机数，并且以后也不能给它赋值了。

代码第15行和第16行，a1定义为const int*类型，注意这里的const在int*的左侧，它是用修饰指针所指向的变量，即指针指向为常量。第 15 行中把 a1 指向变量 c是允许的，因为这修改的是指针a1本身。但是第16行改变a1指向的内容是不允许的。编译器给出的错误同样是“l-value specifies const object”。

代码第18行和第19行，a2定义为int* const类型，注意这里的const在int*的右侧，它是用修饰指针本身，即指针本身为常量。因此第 18 行中修改指针 a2 本身是不允许的。而第19行修改a2指向的内容是允许的。

代码第21 行和第22 行，a3 定义为const int* const类型，这里有两个const，分别出现在int*的左右两侧，因此它表示不仅指针本身不能修改，并且其指向的内容也不能修改。所以代码第21行和第22行都会出现编译错误。

注意：变量x、a2和a3在声明的同时一定要初始化，因为它们在后面都不能被赋值。而变量a1可以在声明的时候不初始化。

【答案】

代码第13、16、18、21和22行会出现编译错误。

面试题8 说明const与#define的特点及区别

考点：对const与#define的特点及区别的理解

出现频率：★★★

【解析】

#define只是用来做文本替换的。例如。

```
1    #define PI 3.1415926
2    float angel;
3    angel = 30 * PI / 180;
```

那么，当程序进行编译的时候，编译器会首先将“#define Pi 3.1415926”以后所有代码中的“PI”全部换成“3.1415926”，然后进行编译。因此，#define常量则是一个Compile-Time概念，它的生命周期止于编译期，它存在于程序的代码段，在实际程序中它只是一个常数、一个命令中的参数，并没有实际的存在。

const常量存在于程序的数据段，并在堆栈分配了空间。const常量是一个Run-Time的概念，它在程序中确确实实地存在着并可以被调用、传递。const常量有数据类型，而宏常量没有数据类型。编译器可以对const常量进行类型安全检查。

面试题9 C++中const有什么作用（至少说出3个）

考点：对C++中const的理解

出现频率：★★★★★

【解析】

（1）const用于定义常量：const定义的常量编译器可以对其进行数据静态类型安全检查。

（2）const修饰函数形式参数：当输入参数为用户自定义类型和抽象数据类型时，应该将“值传递”改为“const &传递”，可以提高效率。比较下面两段代码：

```
1    void fun(A a);
2    void fun(A const &a);
```

第一个函数效率低。函数体内产生A类型的临时对象用于复制参数a，临时对象的构造、复制、析构过程都将消耗时间。而第二个函数提高了效率。用“引用传递”不需要产生临时对象，节省了临时对象的构造、复制、析构过程消耗的时间。但光用引用有可能改变a，所以加const。

（3）const修饰函数的返回值：如给“指针传递”的函数返回值加const，则返回值不能被直接修改，且该返回值只能被赋值给加const修饰的同类型指针。例如。

```
1    const char *GetChar(void){};
2    char *ch = GetChar(); // error
3    const char *ch = GetChar(); // correct
```

（4）const修饰类的成员函数（函数定义体）：任何不会修改数据

成员的函数都应用const修饰，这样，当不小心修改了数据成员或调用了非const成员函数时，编译器都会报错。const修饰类的成员函数形式为：

```
1    int GetCount(void)    const;
```

面试题10 static有什么作用（至少说出2个）

考点：对C++中static的作用的理解

出现频率：★★★★★

【解析】

在C语言中，关键字static有3个明显的作用：

（1）在函数体，一个被声明为静态的变量在这一函数被调用的过程中维持其值不变。

（2）在模块内（但在函数体外），一个被声明为静态的变量可以被模块内所有函数访问，但不能被模块外其他函数访问。它是一个本地的全局变量。

（3）在模块内，一个被声明为静态的函数只可被这一模块内的其他函数调用。那就是这个函数被限制在声明它的模块的本地范围内使用。

大多数应试者能正确回答第一部分，而另一部分应试者能正确回答第二部分，但是仅有很少的人能懂得第三部分。作为一名合格的软件工程师，我们要懂得第三部分的作用，要懂得本地化数据和代码范围的好处和重要性。

面试题11 static全局变量与普通的全局变量有什么区别

static 全局变量与普通的全局变量有什么区别？static 局部变量和普通的局部变量有什么区别？static函数与普通函数有什么区别？

考点：对C++中static的作用的理解

出现频率：★★★★

【解析】

全局变量的说明之前再加上static就构成了静态的全局变量。全局变量本身就是静态存储方式，静态全局变量当然也是静态存储方式。这两者在存储方式上并无不同。这两者的区别在于，非静态全局变量的作用域是整个源程序，当一个源程序由多个源文件组成时，非静态的全局变量在各个源文件中都是有效的；而静态全局变量则限制了其作用域，即只在定义该变量的源文件内有效，在同一源程序的其他源文件中不能使用它。由于静态全局变量的作用域局限于一个源文件内，只能为该源文件内的函数公用，因此可以避免在其他源文件中引起错误。

从以上分析可以看出，把局部变量改变为静态变量后是改变了它的存储方式，即改变了它的生存期；把全局变量改变为静态变量后是改变了它的作用域，限制了它的使用范围。

static 函数与普通函数的作用域不同。static函数的作用域仅在本文件，只在当前源文件中使用的函数应该说明为内部函数（static），内部函数应该在当前源文件中说明和定义。对于可在当前源文件以外使用的函数，应该在一个头文件中说明，要使用这些函数的源文件要包含这个头文件。

因此，static全局变量与普通的全局变量的区别是，static全局变量

只初始化一次，防止在其他文件单元中被引用；static局部变量和普通局部变量的区别是，static局部变量只被初始化一次，下一次依据上一次结果值；static函数与普通函数的区别是，static函数在内存中只有一份，普通函数在每个被调用中维持一份复制品。

【答案】

static全局变量与普通全局变量的区别是，static全局变量只初始化一次，防止在其他文件单元中被引用。

static局部变量和普通局部变量的区别是，static局部变量只被初始化一次，下一次依据上一次结果值。

static函数与普通函数的区别是，static函数在内存中只有一份，普通函数在每个被调用中维持一份复制品。

面试题12 看代码写结果——C++类的静态成员

考点：对C++中类的静态成员的理解

出现频率：★★★★★

```
1    #include <iostream.h>
2    class widget
3    {
4    public:
5        widget()
6        {
7            count++;
8        }
9    ～widget()
10       {
11           ——count;
12       }
13       static int num()
14       {
15           return count;
16       }
17   private:
18       static int count;
19   };
20
```

```
21  int widget::coun t = 0;
22
23  int main()
24  {
25      widget x,y;
26      cout << "The Num.is" << widget::num() << endl;
27      if(widget::num()>1)
28      {
29          widget x, y, z;
30          cout << "The Num.is" << widget::num() << endl;
31      }
32      widget z;
33      cout << "The Num.is" << widget::num() << endl;
34
35      return 0;
36  }
```

【解析】

类widget有一个静态成员count和一个静态方法num()。我们知道，类中的静态成员或方法不属于类的实例，而属于类本身并在所有类的实例间共享。在调用它们时应该用类名加上操作符“::”来引用。

在代码第7行类widget的构造方法里把静态成员count的值加1，在代码第11行类widget的析构方法里把静态成员count的值减1。也就是说，静态成员count的值表示类widget实例的个数。

通过以上的分析，可以看到运行到代码第26行时，只有两个类widget的实例，运行到代码第30行时，又产生了3个实例，然后这3个实例在第31行结束后被销毁。最后运行到代码第32行又产生了1个实例。

【答案】

The Num.is2

The Num.is5

The Num.is3

面试题13 使用sizeof计算普通变量所占空间大小

考点：使用sizeof计算普通变量所占空间大小

出现频率：★★★★

假设在32位WinNT操作系统环境下，代码如下。

```
1    char str[] = "Hello";
2    char *p = str;
3    int n = 10 ;
4    sizeof(str) = ______;
5    sizeof(p) = ______;
6    sizeof(n) = _______;
7    void Func ( char str[100] )
8    {
9        sizeof(str) = ____;
10    }
11    void *p = malloc(100);
12    sizeof(p) =______;
```

【解析】

代码第 4 行，str 变量表示数组，对数组变量做 sizeof 运算得到的是数组占用内存的总大小。在这里，str的总大小为strlen("Hello")+1，注意数组中要有一个元素保存字符串结束符，所以sizeof(str)为6。

代码第5行和第6行中的p和n分别是指针和int型变量。在32位WinNT平台下，指针和int都是4个字节，所以结果都是4。

代码第9 行中的str 是函数的参数，它在做sizeof 运算时被认为是指

针。这是因为当我们调用函数 Func（str）时，由于数组是“传址”的，程序会在栈上分配一个4字节的指针来指向数组，因此结果也是4。

代码第11 行中的p 首先指向一个100 字节的堆内存。这里还是对指针做sizeof 运算，结果仍然是4。

总之，数组和指针的 sizeof 运算有细微的区别。如果数组变量被传入函数中做 sizeof运算，则和指针的运算没有区别，否则得到整个数组占用内存的总大小。对于指针，无论是何种类型的指针，其大小都是固定的，在32位WinNT平台下都是4。

面试题14 使用sizeof计算类对象所占空间大小

考点：使用sizeof计算类对象所占空间大小

出现频率：★★★★

请指出下面程序的输出是什么（在32位WinNT操作系统环境下）。

```
1    #include <iostream.h>
2
3    class A
4    {
5    public:
6        int i;
7    };
8
9    class B
10   {
11   public:
12       char ch;
13   };
14
15   class C
16   {
17   public:
18       int i;
19       short j;
```

```
20  };
21
22  class D
23  {
24  public:
25      int i;
26      short j;
27      char ch;
28  };
29
30  class E
31  {
32  public:
33      int i;
34      int ii;
35      short j;
36      char ch;
37      char chr;
38  };
39
40  class F
41  {
42  public:
43      int i;
44      int ii;
45      int iii;
46      short j;
```

```
47      char ch;
48      char chr;
49  };
50
51  int main()
52  {
53      cout << "sizeof(int) = " << sizeof(int) << endl;
54      cout << "sizeof(short) = " << sizeof(short) << endl;
55      cout << "sizeof(char) = " << sizeof(char) << endl;
56      cout << endl;
57      cout << "sizeof(A) = " << sizeof(A) << endl;
58      cout << "sizeof(B) = " << sizeof(B) << endl;
59      cout << "sizeof(C) = " << sizeof(C) << endl;
60      cout << "sizeof(D) = " << sizeof(D) << endl;
61      cout << "sizeof(E) = " << sizeof(E) << endl;
62      cout << "sizeof(F) = " << sizeof(F) << endl;
63
64      return 0;
65  }
```

【解析】

这里定义了6个类，很多人遇到对结构体或类做sizeof运算时，只是简单地把各个成员所占的内存数量相加。在32位WinNT操作系统环境下，char占1个字节，int占4个字节， short占2个字节。因此会很轻易地做出以下计算。

sizeof(A) = sizeof(int) = 4

sizeof(B) = sizeof(char) = 1

sizeof(C) = sizeof(int) + sizeof(short) = 4 + 2 = 6

sizeof(D) = sizeof(int) + sizeof(short) + sizeof(char) = 4 + 2 + 1 = 7

sizeof(E) = 2 * sizeof(int) + 2 * sizeof(char) + sizeof(short) = 2*4 + 2*1 + 2 = 12

sizeof(F) = 3 * sizeof(int) + 2 * sizeof(char) + sizeof(short) = 3*4 + 2*1 + 2 = 15

可实际上，程序执行的结果如下。

sizeof(A) = 4

sizeof(B) = 1

sizeof(C) = 8

sizeof(D) = 8

sizeof(E) = 12

sizeof(F) = 16

这个问题会使许多初学者疑惑不解，这都是由于字节对齐引起的。对齐的作用和原因：各个硬件平台对存储空间的处理上有很大的不同。一些平台对某些特定类型的数据只能从某些特定地址开始存取。其他平台可能没有这种情况，但是最常见的是，如果不按照适合其平台的要求对数据存放进行对齐，会给存取效率带来损失。例如，有些平台每次读都是从偶地址开始，如果一个int型（假设为32位系统）存放在偶地址开始的地方，那么一个读周期就可以读出；而如果存放在奇地址开始的地方，可能会需要 2 个读周期，并对两次读出的结果的高低字节进行拼凑才能得到该int型数据。显然，在读取效率上下降很多。这也是空间和时间的博弈。

对齐的实现：通常，我们写程序的时候，不需要考虑对齐问题。编译器会替我们选择适合目标平台的对齐策略。当然，我们也可以通知给编译器传递预编译指令而改变对指定数据的对齐方法。

字节对齐的细节和编译器实现相关，一般而言，需要满足3个准则：

结构体变量的首地址能够被其最宽基本类型成员的大小所整除；

结构体每个成员相对于结构体首地址的偏移量（offset）都是成员大小的整数倍，如有需要，编译器会在成员之间加上填充字节（internal adding）；

结构体的总大小为结构体最宽基本类型成员大小的整数倍，如有需要，编译器会在最末一个成员之后加上填充字节（trailing padding）。

现在以下面的结构体为例。

```
1  struct S
2  {
3      char c1;
4      int i;
5      char c2;
6  };
```

c1的偏移量为0，i的偏移量为4，c1与i之间便需要3个填充字节。c2的偏移量为8，加起来就是1+3+4+1，等于9个字节。由于这里最宽的基本类型为int，大小为4个字节，再补3个字节凑成4的倍数。这样一共是12个字节。现在给出例题代码中各个类大小的计算过程：

sizeof(A) = 4;

izeof(B) = 1

sizeof(C) = 4 + 1 + 3（补齐）= 8

sizeof(D) = 4 + 2 + 1 + 1（补齐）= 8

sizeof(E) = 4 + 4 + 2 + 1 + 1 = 12

sizeof(F) = 4 + 4 + 4 + 2 + 1 + 1 = 16

【答案】

sizeof(A) = 4

sizeof(B) = 1

sizeof(C) = 8

sizeof(D) = 8

sizeof(E) = 12

sizeof(F) = 16

面试题15 使用sizeof计算含有虚函数的类对象的空间大小

考点：使用sizeof计算含有虚函数的类对象的空间大小

出现频率：★★★★

请指出下面程序的输出是什么（在32位WinNT操作系统环境下）。

```
1    #include <iostream>
2    using namespace std;
3
4    class Base
5    {
6    public:
7        Base(int x) : a(x)
8        {
9        }
10
11       void print()
12       {
13           cout << "base" << endl;
14       }
15   private:
16       int a;
17   };
18
19   class Derived : public Base
```

```
20  {
21  public:
22      Derived(int x) : Base(x-1), b(x)
23      {
24      }
25
26      void print()
27      {
28          cout << "derived" << endl;
29      }
30  private:
31      int b;
32  };
33
34  class A
35  {
36  public:
37      A(int x) : a(x)
38      {
39      }
40      virtual void print()
41      {
42          cout << "A" << endl;
43      }
44  private:
45      int a;
46  };
```

```
47
48  class B : public A
49  {
50  public:
51      B(int x) : A(x-1), b(x)
52      {
53      }
54      virtual void print()
55      {
56          cout << "B" << endl;
57      }
58  private:
59      int b;
60  };
61
62  int main()
63  {
64      Base obj1(1);
65      cout << "size of Base obj is " << sizeof(obj1) << endl;
66      Derived obj2(2);
67      cout << "size of Derived obj is " << sizeof(obj2) << endl;
68
69      A a(1);
70      cout << "size of A obj is " << sizeof(a) << endl;
71      B b(2);
72      cout << "size of B obj is " << sizeof(b) << endl;
73
```

```
74        return 0;
75    }
```

【解析】

上面这个程序定义了4个类，分别是Base、Derived、A和B。其中Derived类是Base的子类，B是A的子类。下面是各个类的大小分析。

对于Base 类来说，它占用内存大小为sizeof(int)，等于4，print()函数不占内存。

对于Derived类来说，比Base类多一个整型成员，因而多4个字节，一共是12个字节。

对于 A 类来说，由于它含有虚函数，因此占用的内存除了一个整型变量之外，还包括一个隐含的虚表指针成员，一共是8个字节。

对于B 类来说，比A 类多一个整型成员，因而多4 个字节，一共是12 个字节。

通过这个例子可以看出，普通函数不占用内存，只要有虚函数，就会占用一个指针大小的内存，原因是系统多用了一个指针维护这个类的虚函数表，并且注意这个虚函数无论含有多少项（类中含有多少个虚函数）都不会再影响类的大小。

【答案】

size of Base obj is 4

size of Derived obj is 8

size of A obj is 8

size of B obj is 12

面试题16 使用sizeof计算虚拟继承的类对象的空间大小

考点：使用sizeof计算虚拟继承的类对象的空间大小

出现频率：★★★★

```
1   #include <iostream>
2   using namespace std;
3
4   class A
5   {
6   };
7
8   class B
9   {
10   };
11
12   class C:public A, public B
13   {
14   };
15
16   class D:virtual public A
17   {
18   };
19
20   class E:virtual public A, virtual public B
```

```
21  {
22  };
23
24  class F
25  {
26  public:
27      int a;
28      static int b;
29  }
30
31  int F::b = 10;
32
33  int main()
34  {
35      cout << "sizeof(A) = " << sizeof(A) << endl;
36      cout << "sizeof(B) = " << sizeof(B) << endl;
37      cout << "sizeof(C) = " << sizeof(C) << endl;
38      cout << "sizeof(D) = " << sizeof(D) << endl;
39      cout << "sizeof(E) = " << sizeof(E) << endl;
40      cout << "sizeof(F) = " << sizeof(F) << endl;
41
42      return 0;
43  }
```

【解析】

程序说明如下。

代码第 35 行，由于 A 是空类，编译器会安插一个 char 给空类，用来标记它的每一个对象。因此其大小为1个字节。

代码第36 行，类B 大小和A 相同，都是1 个字节。

代码第37 行，类C 是多重继承自A 和B，其大小仍然为1 个字节。

代码第38 行，类D 是虚继承自A，编译器为该类安插一个指向父类的指针，指针大小为4。由于此类有了指针，编译器不会安插一个char了。因此其大小是4个字节。

代码第 39 行，类 E 虚继承自 A 并且也虚继承自 B，因此它有指向父类 A 的指针与父类B的指针，加起来大小为8个字节。

代码第40行，类F含有一个静态成员变量，这个静态成员的空间不在类的实例中，而是像全局变量一样在静态存储区中，被每一个类的实例共享，因此其大小是4个字节。

【答案】

sizeof(A) = 1

sizeof(B) = 1

sizeof(C) = 1

sizeof(D) = 4

sizeof(E) = 8

sizeof(F) = 4

面试题17 sizeof与strlen有哪些区别

考点：理解sizeof与strlen的区别

出现频率：★★★

【解析】

它们的区别如下。

sizeof 是操作符，strlen 是函数。

sizeof 操作符的结果类型是size_t，它在头文件中typedef 为unsignedint 类型，该类型保证能容纳实现所建立的最大对象的字节大小。

sizeof 可以用类型做参数，strlen 只能用char*做参数，且必须是以"\0"结尾的。

数组做sizeof 的参数不退化，传递给strlen 就退化为指针了。

大部分编译程序在编译的时候 sizeof 就被计算过了，这就是sizeof(x)可以用来定义数组维数的原因。strlen的结果要在运行的时候才能计算出来，它用来计算字符串的长度，不是类型占内存的大小。

sizeof 后如果是类型，必须加括弧；如果是变量名，可以不加括弧。这是因为sizeof是个操作符，而不是个函数。

在计算字符串数组的长度上有区别。例如，

```
1    char str[20]="0123456789";
2    int a=strlen(str);
3    int b=sizeof(str);
```

a计算的是以0x00结束的字符串的长度（不包括0x00结束符），这里结果是10。

b计算的则是分配的数组str[20]所占的内存空间的大小，不受里面存

储内容的改变而改变，这里结果是20。

如果要计算指针指向的字符串的长度，则一定要使用strlen。例如。

```
1    char* ss = "0123456789";
2    int a = sizeof(ss);
3    int b = strlen(ss);
```

a计算的是ss指针占用的内存空间大小，这里结果是4。

b计算的是ss指向的字符串的长度，这里结果是10。

面试题18 sizeof有哪些用途

考点：理解sizeof的用途

出现频率：★★★★

【解析】

sizeof有以下用途。

与存储分配和I/O 系统那样的例程进行通信。例如，

```
1    void* malloc(size_t size);
2    size_t fread(void* ptr,size_t size,size_t nmemb, FILE* stream);
```

查看某个类型的对象在内存中所占的单元字节。例如，

```
1    void* memset(void* s,int c, sizeof(s));
```

在动态分配一对象时，可以让系统知道要分配多少内存。

便于一些类型的扩充，在 Windows 中很多结构类型就有一个专用的字段是用来放该类型的字节大小的。

由于操作数的字节数在实现时可能出现变化，建议在涉及操作数字节大小时用sizeof来代替常量计算。

如果操作数是函数中的数组形参或函数类型的形参，则sizeof给出其指针的大小。

面试题19 找错——使用strlen()函数代替sizeof计算字符串长度

考点：sizeof不能用于计算字符串长度

出现频率：★★★

```
1    #include <iostream.h>
2    #include <string.h>
3
4    void UpperCase(char str[])
5    {
6        int test = sizeof(str);
7        int test2 = sizeof(str[0]);
8
9        for(size_t i=0; i<sizeof(str)/sizeof(str[0]); ++i)
10           if('a'<=str[i] && str[i]<='z')
11               str[i] -= ('a'-'A');
12   }
13
14   int main()
15   {
16       char str[] = "aBcDe";
17       cout << "The length of str is " << sizeof(str)/sizeof(str[0]) <<
endl;
18       UpperCase( str );
19       cout << str << endl;
```

```
20      return 0;
21  }
```

【解析】

这个程序存在下面两方面的问题。

（1）函数UpperCase(char str[])的意图是将str 指向的字符串中小写字母转换为大写字母。于是在代码第9行利用sizeof(str)/sizeof(str[0])获得数组中的元素个数以便做循环操作。然而，sizeof(str)得到的并不是数组占用内存的总大小，而是一个字符指针的大小，为4字节。因此这里只能循环4次，在代码第18行main()函数的调用中只能改变对数组的前四个字符进行转换。转换的结果为“ABCDe”。

（2）在代码第 17 行，这里的意图是使用要打印字符串的长度。然而，sizeof(str)/sizeof(str[0])计算的是数组元素的个数，比字符串的长度大1，原因是数组的长度还包括字符串的结束符'\0'。

应该用strlen()函数来代替sizeof计算字符串长度。正确的代码如下。

```
1   #include <iostream.h>
2   #include <string.h>
3
4   void UpperCase(char str[])
5   {
6       int test = sizeof(str);
7       int test2 = sizeof(str[0]);
8
9       for(size_t i=0; i<strlen(str); ++i)    //计算字符串的长度
10          if('a'<=str[i] && str[i]<='z')
11              str[i] -= ('a'-'A');
12  }
```

13

14 int main()

15 {

16 char str[] = "aBcDe";

17

18 cout << "The length of str is " << strlen(str) << endl; //计算字符串的长度

19 UpperCase(str);

20 cout << str << endl;

21 return 0;

22 }

【答案】

代码第9行和第17行中的sizeof(str)/sizeof(str[0])应该用strlen(str)替换。

面试题20 使用sizeof计算联合体的大小

写出以下代码的输出结果。

考点：使用sizeof计算联合体的大小

出现频率：★★★★

```
1    #include <iostream.h>
2
3    union u
4    {
5        double a;
6        int b;
7    };
8
9    union u2
10   {
11       char a[13];
12       int b;
13   };
14
15   union u3
16   {
17       char a[13];
18       char b;
19   };
20
```

```
21  int main()
22  {
23      cout<<sizeof(u)<<endl;
24      cout<<sizeof(u2)<<endl;
25      cout<<sizeof(u3)<<endl;
26
27      return 0;
28  }
```

【解析】

这个程序定义了3个联合体u、u2和u3。我们知道联合体的大小取决于它所有的成员中占用空间最大的一个成员的大小。并且对于复合数据类型，如 union、struct、class 的对齐方式为成员中最大的成员对齐方式。

对于u来说，大小就是最大的double类型成员a了，即 sizeof(u)=sizeof(double)=8。

对于 u2 来说，最大的空间是 char[13]类型的数组。这里要注意，由于它的另一个成员int b 的存在，u2 的对齐方式变成4，也就是说，u2 的大小必须在4的对齐上，所以占用的空间变为最接近13的对齐，即16。

对于u3 来说，最大的空间是char[13]类型的数组，即 sizeof(u3)=13。

这里又出现了CPU对齐的问题。所以编译器会尽量把数据放在它的对齐上以提高内存的命中率。对齐是可以更改的，使用#pragma pack(x)可以改变编译器的对齐方式。C++固有类型的对齐取编译器对齐方式与自身大小中较小的一个。例如，指定编译器按 2 对齐，int类型的大小是4，则int的对齐为2和4中较小的2。在默认的对齐方式下，因为几乎所有的数据类型都不大于默认的对齐方式8（除了long double），所以所有的固有类型的对齐方式可以认为就是类型自身的大小。例如下面的程

序：

```
1   #include <iostream.h>
2
3   #pragma pack(2)
4
5   union u
6   {
7       char buf[9];
8       int a;
9   };
10
11   int main()
12   {
13       cout << sizeof(u) << endl;
14
15       return 0;
16   }
```

C++固有类型的对齐取编译器对齐方式与自身大小中较小的一个。

上面的程序中，由于使用手动更改对齐方式为2，所以int的对齐也变成了2（int自身对齐为4），u的对齐取成员中最大的对齐，也是2，所以此时sizeof(u)=10。

如果注释上面的第3行，int的对齐使用4，u的对齐取成员中最大的对齐，也是4，所以此时sizeof(u)=12。

【答案】

8

16

13

面试题21 #pragma pack的作用

选择代码输出的正确结果。

考点：理解#pragma pack 指令的作用

出现频率：★★★★

```
1    #include <iostream.h>
2
3    #pragma pack(1)
4
5    struct test {
6            char c;
7            short s1;
8            short s2;
9            int i;
10   };
11
12   int main()
13   {
14       cout<< sizeof(test) <<endl;
15
16       return 0;
17   }
```

A．9 B．10 C．12 D．16

【解析】

代码第3行用#pragma pack将对齐设为1。由于结构体test中的成员

s1、s2和i的自身对齐分别为2、2和4，都小于1。因此它们都是用1作为对齐，sizeof(test)=1+2+2+4=9。

如果注释代码第3行，则编译器默认对齐为8。所以各个成员自身的对齐都小于8，因此它们使用自身的对齐，sizeof(test)=1+1（补齐）+ 2 + 2 + 2（补齐）+ 4 = 12。

【答案】

A

面试题22 为什么要引入内联函数

考点：理解内联函数的作用

出现频率：★★★★

【解析】

引入内联函数的主要目的是，用它替代 C 语言中表达式形式的宏定义来解决程序中函数调用的效率问题。在C语言里可以使用如下的宏定义。

```
1    #define ExpressionName(Var1,Var2) (Var1+Var2)*(Var1-Var2)
```

这种宏定义在形式及使用上像一个函数，但它使用预处理器实现，没有了参数压栈、代码生成等一系列的操作，因此效率很高。这种宏定义在形式上类似于一个函数，但在使用它时，仅仅只是做预处理器符号表中的简单替换，因此它不能进行参数有效性的检测，也就不能享受C++编译器严格类型检查的好处。另外，它的返回值也不能被强制转换为可转换的合适类型，这样，它的使用就存在着一系列的隐患和局限性。

另外，在 C++中引入了类及类的访问控制，这样，如果一个操作或者说一个表达式涉及类的保护成员或私有成员，你就不可能使用这种宏定义来实现（因为无法将 this 指针放在合适的位置）。

inline推出的目的，也正是为了取代这种表达式形式的宏定义。它消除了它的缺点，同时又很好地继承了它的优点。

面试题23 为什么inline能很好地取代表达式形式的预定义

考点：理解内联函数相比于宏定义的优越之处

出现频率：★★★

【解析】

有如下几种原因：

inline定义的类的内联函数，函数的代码被放入符号表中，在使用时直接进行替换（像宏一样展开），没有了调用的开销，效率也很高。

类的内联函数也是一个真正的函数。编译器在调用一个内联函数时，首先会检查它的参数的类型，保证调用正确；然后进行一系列的相关检查，就像对待任何一个真正的函数一样。这样就消除了它的隐患和局限性。

inline 可以作为某个类的成员函数，当然就可以在其中使用所在类的保护成员及私有成员。

面试题24 说明内联函数使用的场合

考点：理解内联函数的作用场合

出现频率：★★★

【解析】

首先使用inline函数可以完全取代表达式形式的宏定义。

内联函数在 C++类中应用最广的，应该是用来定义存取函数。我们定义的类中一般会把数据成员定义成私有的或者保护的，这样，外界就不能直接读写我们类成员的数据了。对于私有或者保护成员的读写就必须使用成员接口函数来进行。如果我们把这些读写成员函数定义成内联函数的话，将会获得比较好的效率。例如下面的代码：

```
1   Class A
2   {
3   Private:
4           int nTest;
5   Public:
6           int readTest()
7           {
8               return nTest;
9           }
10      void setTest(int i);
11   };
12
13   inline void A::setTest(int i)
14   {
```

```
15      nTest = i;
16   };
```

类 A 的成员函数 readTest()和 setTest()都是 inline 函数。readTest()函数的定义体被放在类声明之中，因而readTest()自动转换成inline函数；setTest()函数的定义体在类声明之外，因此要加上inline关键字。

面试题25 为什么不把所有的函数都定义成内联函数

考点：理解内联函数的缺点

出现频率：★★★★

【解析】

内联是以代码膨胀（复制）为代价的，仅仅省去了函数调用的开销，从而提高函数的执行效率。如果执行函数体内代码的时间相比于函数调用的开销较大，那么效率的收获会很少。另一方面，每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。以下情况不宜使用内联。

如果函数体内的代码比较长，使用内联将导致内存消耗代价较高。

如果函数体内出现循环，那么执行函数体内代码的时间要比函数调用的开销大。

另外，类的构造函数和析构函数容易让人误解成使用内联更有效。要当心构造函数和析构函数可能会隐藏一些行为，如“偷偷地”执行了基类或成员对象的构造函数和析构函数。

所以不要随便地将构造函数和析构函数的定义体放在类声明中。一个好的编译器将会根据函数的定义体，自动地取消不值得的内联（这说明了 inline 不应该出现在函数的声明中）。

面试题26 内联函数与宏有什么区别

考点：理解内联函数与宏定义的区别

出现频率：★★★★

【解析】

二者区别如下：

内联函数在编译时展开，宏在预编译时展开。

在编译的时候，内联函数可以直接被镶嵌到目标代码中，而宏只是一个简单的文本替换。

内联函数可以完成诸如类型检测、语句是否正确等编译功能，宏就不具有这样的功能。

宏不是函数，inline 函数是函数。

宏在定义时要小心处理宏参数（一般情况是把参数用括号括起来），否则容易出现二义性。而内联函数定义时不会出现二义性。

第3章 引用和指针

引用是C++引入的新语言特性，是C++常用的一个重要内容。正确、灵活地使用引用，可以使程序简洁、高效。

指针是C语言中广泛使用的一种数据类型。运用指针编程是C语言最主要的风格之一。使用指针可以编写出精炼而高效的程序。学习指针是学习 C 语言中最重要的一环，同时，指针也是学习C语言中最为困难的一部分。

指针可以表示各种数据对象，例如简单变量、数组、结构体、类以及函数等。指针具有不同的类型，这些类型指向不同的数据存储体。

在各种笔试和面试中，许多指针方面的问题都是各个公司的重点考点，例如数组指针、函数指针、常量指针、指针传值、多维指针等。

本章通过许多实际公司面试题对指针的各个方面的重点、难点进行全面且细致的分析。通过阅读本章，希望读者能解决指针的各个难点。

面试题1 一般变量引用

仔细阅读下面的代码，并分析变量的结果和程序运行结果。

考点：一般变量引用

出现频率：★★★★

```
1    #include <iostream>
2    #include <string>
3    using namespace std;
4
5    int main(int argc, char* argv[])
6    {
7        int a = 10;
8        int b = 20;
9        int &rn = a;
10         int equal;
11
12         rn=b;
13         cout << "a = " << a << endl;
14         cout << "b = " << b << endl;
15
16         rn = 100;
17
18         cout << "a = " << a << endl;
19         cout << "b = " << b << endl;
20
```

```
21      equal = (&a == &rn)? 1: 0;
22
23      cout << "equal = " << equal << endl;
24
25      return 0;
26  }
```

【解析】

代码第7 行和第8 行，整型变量a 和整型变量b 分别被初始化为10 和20。

代码第9 行，声明rn 为变量a 的一个引用。

代码第12 行，将rn 的值赋为b 的值。此时rn 其实就是a 的一个别名，对rn 的赋值其实就是对a的赋值。因此执行完赋值后，a的值就是b的值，即都是20。

代码第16 行，将rn 的值赋为100，于是a 的值变成了100。

代码第21行，将a的地址与rn的地址进行比较，如果相等，变量equal的值为1，否则为0。将rn声明为a的引用，不需要为rn另外开辟内存单元。rn和a占内存中的同一个存储单元，它们具有同一地址，所以equal为1。

【答案】

```
1   a = 20
2   b = 20
3   a = 100
4   b = 20
5   equal = 1
```

面试题2 指针变量引用

仔细阅读下面的代码，并分析变量的结果和程序运行结果。

考点：指针变量引用

出现频率：★★★★

```
1    #include <iostream>
2    using namespace std;
3    int main(int argc, char* argv[])
4    {
5        int a = 1;
6        int b = 10;
7        int* p = &a;
8        int* &pa = p;
9
10        (*pa)++;
11        cout << "a = " << a << endl;
12        cout << "b = " << b <<endl;
13        cout << "*p = " << *p <<endl;
14
15        pa = &b;
16        (*pa)++;
17        cout << "a = " << a << endl;
18        cout << "b = " << b <<endl;
19        cout << "*p = " << *p <<endl;
20
```

```
21        return 0;
22    }
```

【解析】

代码第5 行和第6 行，整型变量a 和整型变量b 分别被初始化为1 和10。

代码第7 行，声明整型的指针变量p 并初始指向a。

代码第8 行，声明p 的一个指针引用pa。

代码第10 行，将pa 指向的内容加1。由于pa是p的引用，所以此时实际上是对p指向的内容加1，也就是a加1，结果为a变成了2。

代码第15 行，将pa 指向变量b 的地址。由于pa 是p 的引用，所以此时p 也指向了b的地址。

代码第16 行，将pa 指向的内容加1。由于pa是p的引用，所以此时实际上是对p指向的内容加1，也就是b加1，结果为b变成了12。

【答案】

```
1    a = 2
2    b = 10
3    *p = 2
4    a = 2
5    b = 11
6    *p = 11
```

面试题3 看代码找错误——变量引用

考点：一般变量引用

出现频率：★★★

```
1    #include <iostream>
2    using namespace std;
3
4    int main(int argc, char* argv[])
5    {
6        int a = 1, b = 2;
7        int &c;
8        int &d = a;
9        &d = b;
10       int *p;
11
12       *p = 5;
13
14       return 0;
15   }
```

【解析】

代码第6 行，正确，声明并初始化整型变量a 和b。

代码第7 行，编译错误。声明了一个引用类型的变量c，但是没有初始化。这里会出现编译错误，因为引用类型的变量在声明的同时必须初始化。

代码第8 行，正确。声明了一个变量a 的引用d。

代码第 9 行，编译错误，把 d 作为变量 b 的别名。这是因为引用只能在声明的时候被赋值，以后都不能再把该引用名作为其他变量名的别名。

代码第10 行，正确。声明了一个整型的指针变量p。

代码第12 行，运行错误，把p 的内容赋为5。由于p 没有被初始化，因此此时的p是个野指针，对野指针的内容赋值是非常危险的，会导致程序运行时崩溃。

面试题4 如何交换两个字符串

考点：参数引用

出现频率：★★★

【解析】

源代码如下：

```
1    #include<iostream.h>
2    #include<string.h>
3    void swap(char *&x, char *&y)
4    {
5        char *temp;
6        temp=x;
7        x=y;
8        y=temp;
9    }
10
11   int main()
12   {
13       char *ap = "hello";
14       char *bp = "how are you?";
15
16       cout << "ap:" << ap << endl;
17       cout << "bp:" << bp << endl;
18
19       swap(ap, bp);
```

```
20
21      cout << "swap ap,bp" << endl;
22      cout << "ap:" << ap << endl;
23      cout << "bp:" << bp << endl;
24
25      return 0;
26  }
```

这里swap函数是利用传指针引用实现字符串交换的。由于swap函数是指针引用类型，因此传入函数的就是实参，而不是形参。

如果不用指针引用，那么指针交换只能在swap函数中有效，因为在函数体中，函数栈会分配两个临时的指针变量分别指向两个指针参数，对实际的ap和bp没有影响。

函数执行的结果为：

```
1  ap:hello
2  bp:how are you?
3  swap ap,bp
4  ap:how are you?
5  bp:hello
```

从执行结果来看，swap函数确实起到了交换两个字符串的目的。

当然，如果不用引用，还可以用二维指针达到同样的目的。可以把swap函数的定义改为下面的形式。

```
1  void swap1(char **x, char **y)
2  {
3     char *temp;
4     temp = *x;
5        *x = *y;
6        *y = temp;
```

```
7    }
```

并且把源代码第19行改成：

```
1    swap1(&ap, &bp);
```

用这种传指针地址的方式，同样可以起到交换两个字符串的目的。

面试题5 程序查错——参数引用

考点：参数引用

出现频率：★★★

```
1    #include <iostream.h>
2
3    const float pi=3.14f;
4    float f;
5
6    float f1(float r) {
7            f = r*r*pi;
8
9            return f;
10   }
11   float& f2(float r){
12         f = r*r*pi;
13
14         return f;
15   }
16
17   int    main(){
18         float f1(float=5);
19         float& f2(float=5);
20         float a=f1();
21         float& b=f1();
```

```
22        float c=f2();
23        float& d=f2();
24
25        d += 1.0f;
26
27        cout << "a = " << a << endl;
28        cout << "b = " << b << endl;
29        cout << "c = " << c << endl;
30        cout << "d = " << d << endl;
31        cout << "f = " << f << endl;
32
33        return 0;
34    }
```

【解析】

这里f1()函数返回的是全局变量f的值，f2()函数返回的是全局变量f的引用。

代码第18 行，正确。声明函数f1()的默认参数调用，其默认参数值为5。

代码第19 行，正确。声明函数f2()的默认参数调用，其默认参数值为5。

代码第20 行，正确。将变量a 赋为f1()的返回值。

代码第21 行，错误。将变量b 赋为f1()的返回值。因为在f1()函数里，全局变量f的值78.5赋给一个临时变量temp，这个temp变量由编译器隐式地建立，然后建立这个temp的引用b。这里对一个临时变量temp进行引用会发生错误。

代码第 22 行，正确。f2()函数在返回值时并没有隐式地建立临时变量 temp，而是直接将全局变量f返回给主函数。

代码第23行，正确。主函数中都不使用定义变量，而是直接使用全局变量的引用，这种方式是全部4种中最节省内存空间的。但必须注意它所引用的变量的有效期，此处全局变量f的有效期肯定长于引用d，所以是安全的。否则，会出现错误。例如，将一个局部变量的引用返回，此时全局变量f的值为78.5。

代码第25 行，正确。将d 的值加1.0，此时d 是全局变量f 的引用，因此f 的值变成79.5。

【答案】

代码第21行错误。注释第21行与第28行后，运行结果如下：

```
1    A = 78.5
2    C = 78.5
3    D = 79.5
4    F = 79.5
```

面试题6 参数引用的常见错误

挑出下面代码中的错误。

考点：参数引用

出现频率：★★★

```
1    #include <iostream>
2    using namespace std;
3
4    class Test
5    {
6    public:
7        void f(const int& arg);
8    private:
9        int value;
10   };
11
12   void Test::f(const int& arg) {
13       arg = 10;
14       cout << "arg = " << arg << endl;
15       value = 20;
16   }
17
18   int main()
19   {
20       int a = 7;
```

```
21        const int b = 10;
22        int &c = b;
23        const int &d = a;
24
25        a++;
26        d++;
27
28        Test test;
29        test.f(a);
30        cout << "a = " << a << endl;
31
32        return 0;
33    }
```

【解析】

把const放在引用之前表示声明的是一个常量引用。不能使用常量引用修改引用的变量的值。

代码第 13 行，错误。因为参数 arg 是一个常量引用类型的，所以 arg 的值在函数体内不能被修改。

代码第20 行和第21 行，a 被声明为整型变量，b 被声明为整型常量。

代码第22行，声明c为b的引用，错误。其原因是b为常量，而c不是常量引用。

正确的方式应为：

const int &c = b;

代码第23 行，声明d 为a 的常量引用，正确。

代码第25 行，变量a 自增1，正确。

代码第26 行，d 自增1，错误。d 是常量引用，不能对d 使用赋值操

作。

从上面的分析可以看到，对于常量类型的变量，其引用也必须是常量类型的；对于非常量类型的变量，其引用可以是非常量的，也可以是常量的。但是要注意，无论什么情况，都不能使用常量引用修改其引用的变量的值。

【答案】

代码第13行，错误。原因是arg的值不能被修改。

代码第22行，错误。原因是不能使用常量类型变量定义非常量的引用。

代码第26行，错误。原因是不能使用常量引用修改变量的值。

面试题7 指针和引用有什么区别

考点：引用和指针的区别

出现频率：★★★★★

【解析】

区别如下：

（1）初始化要求不同。引用在创建的同时必须初始化，即引用到一个有效的对象；而指针在定义的时候不必初始化，可以在定义后面的任何地方重新赋值。

（2）可修改性不同。引用一旦被初始化为指向一个对象，它就不能被改变为另一个对象的引用；而指针在任何时候都可以改变为指向另一个对象。给引用赋值并不是改变它和原始对象的绑定关系。

（3）不存在 NULL 引用，引用不能使用指向空值的引用，它必须总是指向某个对象；而指针则可以是NULL，不需要总是指向某些对象，可以把指针指向任意的对象，所以指针更加灵活，也容易出错。

（4）测试需要的区别。由于引用不会指向空值，这意味着使用引用之前不需要测试它的合法性；而指针则需要经常进行测试。因此使用引用的代码效率比使用指针的要高。

（5）应用的区别。如果是指一旦指向一个对象后就不会改变指向，那么应该使用引用。如果有存在指向 NULL（不指向任何对象）或在不同的时刻指向不同的对象这些可能性，应该使用指针。

实际上，在语言层面，引用的用法和对象一样；在二进制层面，引用一般都是通过指针来实现的，只不过编译器帮我们完成了转换。总体来说，引用既具有指针的效率，又具有变量使用的方便性和直观性。

面试题8 为什么传引用比传指针安全

考点：引用和指针的区别

出现频率：★★★★

【解析】

由于不存在空引用，并且引用一旦被初始化为指向一个对象，它就不能被改变为另一个对象的引用，因此引用很安全。

对于指针来说，它可以随时指向别的对象，并且可以不被初始化，或为NULL，所以不安全。const 指针仍然存在空指针，并且有可能产生野指针。

面试题9 复杂指针的声明

考点：复杂指针的声明

出现频率：★★★★★

用变量a给出下面的定义：

a．一个整型数（An integer）

b．一个指向整型数的指针（A pointer to an integer）

c．一个指向指针的指针，它指向的指针是指向一个整型数的（A pointer to a pointer to aninteger）

d．一个有10 个整型数的数组（An array of 10 integers）

e．一个有10 个指针的数组，该指针是指向一个整型数的（An array of 10 pointers tointegers）

f．一个指向有10 个整型数数组的指针（A pointer to an array of 10 integers）

g．一个指向函数的指针，该函数有一个整型参数并返回一个整型数（A pointer to a function that takes an integer as an argument and returns an integer）

h．一个有 10 个指针的数组，该指针指向一个函数，该函数有一个整型参数并返回一个整型数（An array of ten pointers to functions that take an integer argument and return aninteger）

【答案】

a．int a;　　　　　　//An integer

b．int *a;　　　　　//Apointer to an integer

c．int **a;　　　　　//Apointer to a pointer to an integer

d．int a[10];　　　//An arrayof 10 integers

e．int *a[10];　　//An arrayof 10 pointers to integers

f．int (*a)[10];　　　　//Apointer to an arrayof 10 integers

g．int (*a)(int);　　　// A pointer　to a function a　that　takes an integer argument andreturns an integer

h．int (*a[10])(int); // An array of 10 pointers to functions that take an integer argument and return an integer

扩展知识：解读复杂指针声明

使用右左法则：首先从最里面的圆括号看起，然后往右看，再往左看。每当遇到圆括号时，就应该掉转阅读方向。一旦解析完圆括号里面所有的东西，就跳出圆括号。重复这个过程，直到整个声明解析完毕。

这里对这个法则进行一个小小的修正，应该是从未定义的标识符开始阅读，而不是从括号读起。这是因为一个声明里面未定义的标识符只会有一个。

现在通过几个例子来讨论如何运用右左法则解读复杂指针声明。

1　int (*func)(int *p);

首先找到那个未定义的标识符，就是func，它的外面有一对圆括号，而且左边是一个*号，这说明func是一个指针；然后跳出这个圆括号，先看右边，也是一个圆括号，这说明(*func)是一个函数，而 func是一个指向这类函数的指针，就是一个函数指针，这类函数具有int*类型的形参，返回值类型是int。

2　int (*func)(int *p, int (*f)(int*));

func被一对括号包含，且左边有一个*号，说明func是一个指针，跳出括号，右边也有个括号，那么func 是一个指向函数的指针，这类函数具有int *和int (*)(int*)这样的形参，返回值为int类型。再来看一看func的形参int (*f)(int*)，类似前面的解释，f 也是一个函数指针，指向的函数具有int*类型的形参，返回值为int。

3　int (*func[5])(int *p);

func右边是一个[]运算符，说明func是一个具有5个元素的数组；func的左边有一个*，说明 func 的元素是指针。要注意这里的*不是修饰 func 的，而是修饰 func[5]的，原因是[]运算符优先级比*高，func先跟[]结合，因此*修饰的是func[5]。跳出这个括号，看右边，也是一对圆括号，说明 func 数组的元素是函数类型的指针，它所指向的函数具有 int*类型的形参，返回值类型为int。

4　int (*(*func)[5])(int *p);

func被一个圆括号包含，左边又有一个*，那么func是一个指针；跳出括号，右边是一个[]运算符号，说明func是一个指向数组的指针。现在往左看，左边有一个*号，说明这个数组的元素是指针；再跳出括号，右边又有一个括号，说明这个数组的元素是指向函数的指针。总结一下，就是：func 是一个指向数组的指针，这个数组的元素是函数指针，这些指针指向具有int*类型的形参，返回值为int类型的函数。

5　int (*(*func)(int *p))[5];

func是一个函数指针，这类函数具有int*类型的形参，返回值是指向数组的指针，所指向的数组的元素是具有5个int元素的数组。

面试题10 看代码写结果——用指针赋值

考点：用指针赋值

出现频率：★★★★

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       char a[] = "hello, world";
6       char * ptr = a;
7
8       printf("%c\n", *(ptr+4));
9       printf("%c\n", ptr[4]);
10      printf("%c\n", a[4]);
11      printf("%c\n", *(a+4));
12
13      *(ptr+4) += 1;
14      printf("%s\n", a);
15
16      return 0;
17  }
```

【解析】

代码第5 行，声明了一个字符数组a，并初始化为"hello, world"，包括以'\0'结束字符。

代码第 6 行，声明一个字符指针 ptr，并初始化指向数组 a 首（a 的

第一个元素地址）。

代码第8 行，这里将ptr 加4，再输出地址的内容，也就指向了输出a[4]的内容。

代码第9 行，ptr[4]和*(ptr+4)一样，也是a[4]的内容。

代码第10 行，输出a[4]的内容。

代码第11 行，*(a+4)和a[4]一样，也是a[4]的内容。

代码第13 行，使数组a[4]的内容加1。由于原来a[4]的内容为"hello, world"字符串的第五个字符'o'，加1后就是'p'。

【答案】

```
1   o
2   o
3   o
4   o
5   p
```

面试题11 指针加减操作

写出下面程序的结果。

考点：指针加减操作

出现频率：★★★★

```
1    #include <stdio.h>
2
3    int main(void)
4    {
5        int a[5]={1,2,3,4,5};
6        int *ptr=(int *)(&a+1);
7
8        printf("%d\n", *(a+1));
9        printf("%d\n", *(ptr-1));
10
11       return 0;
12   }
```

【解析】

这里主要是考查关于指针加减操作的理解。

对指针进行加1操作，得到的是下一个元素的地址，而不是原有地址值直接加1。所以，一个类型为t的指针的移动，以sizeof(t)为移动单位。

代码第5 行，声明一个一维数组a，并且a 有5 个元素。

代码第6行，ptr是一个int型的指针&a + 1，即取a的地址，该地址的值加sizeof(a)的值，即&a + 5*sizeof(int)，也就是a[5]的地址，显然，当

前指针已经越过了数组的界限。(int *)(&a+1)则是把上一步计算出来的地址，强制转换为int *类型，赋值给ptr。

代码第 8 行，a 与&a 的地址是一样的，但意思不一样。a 是数组首地址，也就是a[0]的地址；&a是对象（数组）首地址，a+1是数组下一元素的地址，即a[1]；而&a+1是下一个对象的地址，即a[5]。因此这里输出为2。

代码第9 行，因为ptr 指向a[5]，并且ptr 是int*类型，所以*(ptr-1)指向a[4]，输出5。

面试题12 指针比较

写出下面程序的结果。

考点：指针比较操作

出现频率：★★★★

```
1    #include <iostream>
2    using namespace std;
3
4    int main(void)
5    {
6        char str1[]       = "abc";
7        char str2[]       = "abc";
8        const char str3[]    = "abc";
9        const char str4[]    = "abc";
10       const char* str5    = "abc";
11       const char* str6    = "abc";
12       char* str7    = "abc";
13       char* str8    = "abc";
14
15       cout << ( str1==str2 ) << endl;
16       cout << ( str3==str4 ) << endl;
17       cout << ( str5==str6 ) << endl;
18       cout << ( str6==str7 ) << endl;
19       cout << ( str7==str8 ) << endl;
20
```

```
21        return 0;
22    }
```

【解析】

这个程序考查的是内存中各个数据的存放方式。

数组str1、str2、str3和str4都是在栈中分配的，内存中的内容都为"abc"加一个'\0'，但是它们的位置是不同的。因此代码第15行和第16行的输出都是0。

指针str5、str6、str7和str8也是在栈中分配的，它们都指向"abc"字符串，注意"abc"存放在数据区，所以str5、str6、str7 和str8 其实指向同一块数据区的内存。因此第17、18和19行的输出是1。

【答案】

```
1    0
2    0
3    1
4    1
5    1
```

面试题13 看代码找错误——内存访问违规

考点：指针操作内存违规

出现频率：★★★★

```
1    #include <iostream>
2    using namespace std;
3
4    int main()
5    {
6        char a;
7        char *str1 = &a;
8        char* str2 = "AAA";
9
10       strcpy(str1, "hello");
11       cout << str1 << endl;
12
13       str2[0]='B';
14       cout << str2 << endl;
15
16       return 0;
17   }
```

【解析】

代码第10 行，str1 指向一个字节大小的内存区。由于复制"hello"字符串最少需要6个字节，显然内存大小不够。因此会因为越界进行内存读写而导致程序崩溃。

代码第 13 行，str2 指向"AAA"这个字符串常量。因为是常量，所以对 str2[0]的赋值操作是不合法的，也会导致程序崩溃。

【答案】

代码第10行导致运行错误。

代码第13行导致运行错误。

面试题14 指针的隐式转换

找出下面代码中的错误。

考点：指针类型的隐式转换

出现频率：★★★★

```
1    #include <stdio.h>
2
3    int main()
4    {
5        int ival = 1024;
6        int ival2 = 2048;
7        int *pi1 = &ival;
8        int *pi2 = &ival2;
9        int **pi3 = 0;
10
11       ival = *pi3;
12       *pi2 = *pi3;
13       ival = pi2;
14       pi2 = *pi1;
15       pi1 = *pi3;
16       ival = *pi1;
17       pi1 = ival;
18       pi3 = &pi2;
19
20       return 0;
```

21　}

【解析】

代码第5 行，正确，声明并初始化整型变量ival。

代码第6 行，正确，声明并初始化整型变量ival2。

代码第7 行，正确，声明整型指针变量pi1，初始化指向ival。

代码第8 行，正确，声明整型指针变量pi2，初始化指向ival2。

代码第9 行，正确，声明二维整型指针变量pi3，初始化为0。

代码第11 行，编译错误，*ival 是int 类型，pi3 是int *类型，不能隐式转换。

代码第12 行，编译错误，*pi2 是int 类型，*pi3 是int *类型，不能隐式转换。

代码第13 行，编译错误，ival 是int 类型，pi2 是int *类型，不能隐式转换。

代码第14 行，编译错误，pi2 是int *类型，*pi1 是int 类型，不能隐式转换。

代码第 15 行，运行时错误，pi3 是 NULL 指针，试图得到*pi3 的值会发生运行错误。

代码第16 行，正确，将ival 的值赋为*pi1。

代码第17 行，编译错误，pi1 是int *类型，ival 是int 类型，不能隐式转换。

代码第18 行，正确，将pi3 的值赋为&pi2，都是int **类型。

本题中错误的类型有两种，一种是编译错误，另一种是运行时错误。导致编译错误的是类型之间不能隐式转换，如 int 转换成 int*、int* 转换成 int 等。导致运行时错误是因为在Windows平台，进程的内存空间有一块是专门用于NULL指针分配的分区，这个分区的地址空间是禁止进入的，因此就会发生内存访问违规现象，同时该进程将终止运行。

【答案】

代码第11、12、13、14、17行编译错误。
第15行运行错误。

面试题15 指针常量与常量指针的区别

考点：指针常量与常量指针的区别

出现频率：★★★★★

【解析】

这里有个小规则，像这样连着的两个词，前面的一个通常是修饰部分，中心词是后面一个词。

常量指针，表述为“是常量的指针”，它首先应该是一个指针。

指针常量，表述为“是指针的常量”，它首先应该是一个常量。

接下来进行详细分析。

常量指针，它是一个指向常量的指针。设置常量指针指向一个常量，为的就是防止写程序过程中对指针误操作出现了修改常量这样的错误，编译系统就会提示我们出错信息。因此，常量指针就是指向常量的指针，指针所指向的地址的内容是不可修改的。

指针常量，它首先是一个常量，然后才是一个指针。指针常量就是不能修改这个指针所指向的地址，一开始初始化指向哪儿，它就只能指向哪儿了，不能指向其他的地方了，就像一个数组的数组名一样，是一个固定的指针，不能对它移动操作。如果使用p++，系统就会提示出错。但是注意，这个指向的地方里的内容是可以替换的，这和上面说的常量指针是完全不同的概念。总之，指针常量就是指针的常量，它是不可改变地址的指针，但是可以对它所指向的内容进行修改。

【答案】

常量指针就是指向常量的指针，它所指向的地址的内容是不可修改的。

指针常量就是指针的常量，它是不可改变地址的指针，但是可以对

它所指向的内容进行修改。

面试题16 指针的区别

考点：const关键字在指针声明时的作用

出现频率：★★★★★

下述4个指针有什么区别？

```
1  char * const p1;
2  char const * p2;
3  const char *p3;
4  const char *constp4;
```

【解析】

如果const位于*号的左侧，则const就是用来修饰指针所指向的变量，即指针指向为常量；如果const位于*号的右侧，const就是修饰指针本身，即指针本身是常量。因此，p1指针本身是常量，但它指向的内容可以被修改。p2和p3的情况相同，都是指针所指向的内容为常量。p4则表示指针本身是常量，并且它指向的内容也不可被修改。

【答案】

p1是指针常量，它本身不能被修改，指向的内容可以被修改。

p2和p3是常量指针，它本身可以被修改，指向的内容不可以被修改。

p4本身是常量，并且它指向的内容也不可被修改。

面试题17 找错——常量指针和指针常量的作用

考点：常量指针和指针常量的作用

出现频率：★★★★

```
1    #include <stdio.h>
2
3    int main()
4    {
5        const char *node1 = "abc";
6        char *const node2 = "abc";
7
8        node1[2] = 'k';
9            *node1[2] = 'k';
10       *node1 = "xyz";
11       node1 = "xyz";
12
13       node2[2] = 'k';
14       *node2[2] = 'k';
15       *node2 = "xyz";
16       node2 = "xyz";
17
18       return 0;
19   }
```

【解析】

上面的代码中，node1和node2分别是常量指针和指针常量，并且都在初始化时指向了常量字符串"abc"。因此，它们对于指向的内存进行修改都是非法的，如果是node1操作，会出现编译错误，而node2会出现运行错误。

【答案】

代码第8、9、10行出现编译错误。

第11行正确。

代码第14、16行出现编译错误，第13、15行出现运行时错误。

面试题18 this指针的正确叙述

考点：this指针的基本概念

出现频率：★★★

下列关于this指针的叙述中，正确的是（　　）。

A．任何与类相关的函数都有this指针

B．类的成员函数都有this指针

C．类的友元函数都有this指针

D．类的非静态成员函数才有this指针

【解析】

A 错误。类的非静态成员函数是属于类的对象，含有this指针。而类的static 函数属于类本身，不含this指针。

B 错误。类的非静态成员函数是属于类的对象，含有this指针。而类的static 函数属于类本身，不含this指针。

C 错误。友元函数是非成员函数，所以它无法通过this 指针获得一份拷贝。

D 正确。

【答案】

D

面试题19 看代码写结果——this指针

考点：this指针的使用

出现频率：★★★

下面的代码输出结果是什么？如果取消第14行的注释，输出又是什么？

```
1    #include <iostream>
2    using namespace std;
3
4    class MyClass
5    {
6    public:
7        int data;
8        MyClass(int data)
9        {
10           this->data = data;
11       }
12       void print()
13       {
14           //cout << data << endl;
15           cout << "hello!" << endl;
16       }
17   };
18
19
```

```
20   int main()
21   {
22       MyClass * pMyClass;
23       pMyClass = new MyClass(1);
24       pMyClass->print();
25       pMyClass[0].print();
26       pMyClass[1].print();
27       pMyClass[10000000].print();
28
29       return 0;
30   }
```

【解析】

这里需要明白类函数是如何被编译以及如何被执行的。

对于类成员函数而言，并不是一个对象对应一个单独的成员函数体，而是此类的所有对象共用这个成员函数体。当程序被编译之后，此成员函数地址即已确定。我们常说，调用类成员函数时，会将当前对象的 this 指针传给成员函数。没错，一个类的成员函数体只有一份，而成员函数之所以能把属于此类的各个对象的数据区别开，就在于每次执行类成员函数时，都会把当前对象的 this 指针（对象首地址）传入成员函数，函数体内所有对类数据成员的访问，都会被转化为this->数据成员的方式。

如果print函数里没有访问对象的任何数据成员，那么此时传进来的对象this指针实际上是没有任何用处的。这样的函数，其特征与全局函数并没有太大区别。但如果取消第 14行的注释，由于print函数要访问类的数据成员data，而类的数据成员是伴随着对象声明而产生的。但是，我们只 new 了一个 MyClass，显然，下标"1"和下标"10000000"的MyClass对象根本不存在，那么对它们的数据成员访问也显然是非法

的。

【答案】

注释代码第14行时的输出：

1　hello!

2　hello!

3　hello!

4　hello!

取消代码第 14行注释后的输出：

1　1

2　hello!

3　1

4　hello!

5　-33686019

6　hello!

7　段错误

面试题20 指针数组与数组指针的区别

考点：指针数组与数组指针的区别

出现频率：★★★★★

【解析】

指针数组指一个数组里存放的都是同一个类型的指针，例如

```
1    int * a[10];
```

数组a 里面存放了10 个int *型变量，由于它是一个数组，已经在栈区分配了10 个（int*）的空间，也就是32位机上是40个byte，每个空间都可以存放一个int型变量的地址。这个时候，你可以为这个数组的每一个元素初始化。

数组指针指一个指向一维或者多维数组的指针，例如

```
int * b=new int[10];
```

指针b指向含有10个整型数据的一维数组。注意，这个时候释放空间一定要delete []，否则会造成内存泄露。

参考下面的源代码：

```
1    #include <iostream>
2    using namespace std;
3
4    int main()
5    {
6        int x1[4] = {1, 2, 3, 4};
7        int x2[2] = {5, 6};
8        int x3[3] = {7, 8, 9};
9        int *a[2];
```

```
10      int *b = x1;
11      int i = 0;
12
13      a[0] = x2;
14      a[1] = x3;
15
16      cout << "输出a[0]: ";
17      for(i = 0; i < sizeof(x2) / sizeof(int); i++)
18      {
19          cout << a[0][i] << " ";
20      }
21      cout << endl;
22
23      cout << "输出a[1]: ";
24      for(i = 0; i < sizeof(x3) / sizeof(int); i++)
25      {
26          cout << a[1][i] << " ";
27      }
28      cout << endl;
29
30      cout << "输出b: ";
31      for (i = 0; i < sizeof(x1) / sizeof(int); i++)
32      {
33          cout << b[i] << " ";
34      }
35      cout << endl;
36
```

```
37        return 0;
38    }
```

这个程序中有指针数组a和数组指针b。a里的两个指针元素分别指向了数组x2和x3，数组指针b指向了数组x1。输出如下：

输出a[0]: 5 6

输出a[1]: 7 8 9

输出b: 1 2 3 4

总之，指针数组表示它是一个数组，并且数组中的每一个元素是指针，而数组指针表示它是一个指针，并且指向了一个数组。

【答案】

指针数组表示它是一个数组，并且数组中的每一个元素是指针。

数组指针表示它是一个指针，并且指向了一个数组。

面试题21 找错——指针数组和数组指针的使用

考点：指针数组和数组指针的使用

出现频率：★★★★

```
1    #include <stdio.h>
2
3    Int main()
4    {
5        char *str[]={"Welcome","to","Fortemedia","Nanjing"};
6        char **p = str + 1;
7        str[0] = (*p++) + 2;
8        str[1] = *(p+1);
9        str[2] = p[1] + 3;
10       str[3] = p[0] + (str[2] - str[1]);
11       printf("%s\n", str[0]);
12       printf("%s\n", str[1]);
13       printf("%s\n", str[2]);
14       printf("%s\n", str[3]);
15
16       return 0;
17   }
```

【解析】

本题的每次执行都较强地依赖于上一个语句执行的情况，好几次一个语句，同时修改str和p的值。

代码第5行结束时，str是下面数组的第一个值。

（1）第1个字符串的首地址的存放地址，标记为A，其内容为"Welcome"。

（2）第2个字符串的首地址的存放地址，标记为B，其内容为"to"。

（3）第3个字符串的首地址的存放地址，标记为C，其内容为"Fortemedia"。

（4）第4个字符串的首地址的存放地址，标记为D，其内容为"Nanjing"。

代码第6行结束时，p指向B。

代码第7行结束时，p指向C。此时str[0]指向第4个字符串"Nanjing"后面的元素，因此其内容为空。

代码第8行结束时，p没有移动，str[1]指向p的后一个元素地址，即D。

代码第9行，此时p[1]指向D。p[1] + 3即指向字符串的元素的第4个元素，即'j'字符。此行执行之后，str[2]等于'j'的地址。

代码第10 行，由第8 行和第9 行可知str[2] – str[1]等于3，而p[0]指向'j'的地址。因此str[4]指向"Nanjing"字符串中的最后一个字符'g'的地址。

【答案】

1　(空)

2　Nanjing

3　jing

4　g

面试题22 函数指针与指针函数的区别

考点：函数指针与指针函数的区别

出现频率：★★★★★

【解析】

指针函数是指带指针的函数，即本质是一个函数，并且返回类型是某一类型的指针。其定义如下：

```
1    返回类型标识符 *返回名称（形式参数表）{函数体 }
```

事实上，每一个函数，即使它不带有返回某种类型的指针，它本身都有一个入口地址，该地址相当于一个指针。比如函数返回一个整型值，实际上也相当于返回一个指针变量的值，不过这时的变量是函数本身而已，而整个函数相当于一个“变量”。

函数指针是指向函数的指针变量，因而它本身首先应是指针变量，只不过该指针变量指向函数。有了指向函数的指针变量后，可用该指针变量调用函数，就如同用指针变量可引用其他类型的变量一样。

请看下面这个例子程序。

```
1    #include <iostream>
2    using namespace std;
3
4    int max(int x,int y)
5    {
6        return(x > y? x: y);
7    };
8
9    float *find(float *p, int x)
```

```
10{
11      return p+x;
12  };
13
14  int main()
15  {
16      float score[] = {10, 20, 30, 40};
17      int (*p)(int, int);
18      float *q = find(score+1, 1);
19      int a;
20
21      p = max;
22      a=(*p)(1, 2);
23
24      cout << "a = " << a << endl;
25      cout << "*q = " << *q << endl;
26
27      return 0;
28  }
```

这里，函数find()被定义为指针函数，指针p被定义为函数指针类型。main函数中调用find()函数时，将数组中第2个元素的地址和偏移量1传入，返回的应该是数组中第3个元素的地址。对于指针p，在第21行被赋为max()函数的地址，因此在第22行使用指针p就能完成调用max()函数的目的。输出如下：

```
a = 2
*q = 30
```

【答案】

指针函数是返回指针类型的函数。
函数指针是指向函数地址的指针。

面试题23 数组指针与函数指针的定义

考点：数组指针与函数指针的定义

出现频率：★★★

定义下面的几种类型变量：

a．含有10个元素的指针数组

b．数组指针

c．函数指针

d．指向函数的指针数组

【答案】

a．int *a[10];

b．int *a = new int[10];

c．void (*fn)(int, int);

d．int (*fnArray[10])(int, int);

面试题24 各种指针的定义

考点：各种指针的定义

出现频率：★★★

写出函数指针、函数返回指针、const指针、指向const的指针、指向const的const指针。

【答案】

void (*f)(int, int)，f 是指向void max(int x, int y)类型的函数指针。

int *fn()，fn 是返回int 指针类型的函数。

const int *p，p 是一个指向const 的指针，指向一个常量。

int* const q，q 是一个const 指针。

const int* const ptr，ptr 是指向const 的const 指针。

面试题25 代码改错——函数指针的使用

考点：函数指针的使用

出现频率：★★★★

下面的程序有什么问题？它打印出3个数的最大者。

```
1    #include <iostream>
2    using namespace std;
3
4    int max(int x, int y)
5    {
6        return x > y? x:y;
7    }
8
9    int main()
10   {
11       int *p;
12       int a, b, c;
13       int result;
14       int max(x, y);
15
16       p = max;
17       cout << "Please input three integer " << endl;
18       cin >> a >> b >> c;
19       result = (*p)((*p)(a, b), c);
20       cout << "result = " << result << endl;
```

```
21
22        return 0;
23    }
```

【解析】

这道程序题中函数指针的使用存在错误。

代码第14行，声明max函数方法错误。

在代码第16行中，p指向max函数地址，这里会出现指针不能转换的错误。p被声明为一个int *类型的指针，但是max 地址却为(int *)(int, int)类型。

【答案】

正确的代码如下：

```
1    #include <iostream>
2    using namespace std;
3
4    int max(int x, int y)
5    {
6        return x > y? x:y;
7    }
8
9    int main()
10   {
11       int (*p)(int, int);              //改正
12       int a, b, c;
13       int result;
14       int max(int, int);               //改正
15
16       p = &max;
```

```
17      cout << "Please input three integer " << endl;
18      cin >> a >> b >> c;
19      result = (*p)((*p)(a, b), c);
20      cout << "result = " << result << endl;
21
22      return 0;
23  }
```

面试题26 看代码写结果——函数指针的使用

考点：函数指针的使用

出现频率：★★★★

```
1    #include <stdio.h>
2    int add1(int a1,int b1);
3    int add2(int a2,int b2);
4    int main(int argc,char* argv[])
5    {
6        int numa1=1,numb1=2;
7        int numa2=2,numb2=3;
8        int (*op[2])(int a,int b);
9        op[0]=add1;
10       op[1]=add2;
11       printf("%d %d\n",op[0](numa1,numb1),op[1](numa2,numb2));
12       getchar();
13
14       return 0;
15   }
16
17   int add1(int a1,int b1)
18   {
19       return a1+b1;
20   }
```

```
21
22  int add2(int a2,int b2)
23  {
24      return a2+b2;
25  }
```

【解析】

在代码第8行，定义了一个函数指针数组op，它含有两个指针元素。在第9行和第10行把这两个元素分别指向了add1和add2两个函数地址。最后在第11行打印出使用函数指针调用add1和add2这两个函数返回的结果。

【答案】

1　3 5

面试题27 typedef用于函数指针定义

考点：函数指针定义中typedef的作用

出现频率：★★★

下面的定义有什么作用？

```
1    typedef int (*pfun)(int x,int y);
```

【解析】

这里的pfun是一个使用typedef自定义的数据类型。它表示一个函数指针，其参数有两个，都是int类型，返回值也是int类型。可以按如下步骤使用：

```
1    typedef int (*pfun)(int x,int y);
2    int fun(int x, int y);
3    pfun p = fun;
4    int ret = p(2, 3);
```

简单说明：

第1 行定义了pfun 类型，表示一个函数指针类型。

第2 行定义了一个函数。

第3 行定义了一个pfun 类型的函数指针p，并赋给它fun 的地址。

第4 行调用p(2, 3)，实现fun(2, 3)的调用功能。

【答案】

定义了一个函数指针类型，表示指向返回值为int，且同时带2个int参数的函数指针类型了。可以用这种类型定义函数指针来调用相同类型的函数。

面试题28 什么是“野指针”

考点：“野指针”是什么以及它的作用

出现频率：★★★★

【解析】

“野指针”不是NULL指针，而是指向“垃圾”内存的指针。人们一般不会错用NULL指针，因为用if语句很容易判断。但是“野指针”是很危险的，if语句对它不起作用。“野指针”的成因主要有两种：

指针变量没有被初始化。任何指针变量刚被创建时不会自动成为NULL 指针，它的默认值是随机的，它会乱指一气。所以，指针变量在创建的同时应当被初始化，要么将指针设置为NULL，要么让它指向合法的内存。

指针p被free或者delete之后，没有置为NULL，让人误以为p是个合法的指针。

【答案】

“野指针”不是NULL指针，而是指向“垃圾”内存的指针。其成因主要为：指针变量没有被初始化，或指针p被free或者delete之后，没有置为NULL。

面试题29 看代码查错——“野指针”的危害

考点：“野指针”的危害

出现频率：★★★★

下面的程序片断有什么重大的bug？

```
1    short *bufptr;
2    short bufarray[20];
3    short var=0x20;
4    *bufptr = var;
5    bufarry[0] = var;
```

【解析】

代码第1 行，正确。声明了一个short *类型的指针，并且没有对它初始化。

代码第2 行，正确。声明了一个20 个元素的数组，每个元素都是short 类型。

代码第3 行，正确。声明了short 类型的变量var，并且把它初始化为0x20。

代码第4 行，错误。将bufptr 指针指向的内容赋为var 变量的值。因为bufptr 没有被初始化，是个“野指针”，因此对它所指向的内容操作是十分危险的，会导致程序崩溃。为了杜绝这种错误，可以将bufptr正确地进行初始化。代码第1行改为：

```
1    short *bufptr = (short *)malloc(sizeof(short));
```

代码第5行，正确。把bufarray的第一个元素赋值为变量var的值。

【答案】

第4行存在重大bug，bufptr是“野指针”，会导致程序运行崩溃。

面试题30 有了malloc/free，为什么还要new/delete

考点：malloc/free和new/delete的区别

出现频率：★★★★

【解析】

malloc与free是C++/C的标准库函数，new/delete是C++的运算符。它们都可用于申请动态内存和释放内存。

对于非内部数据类型的对象而言，光用 malloc/free 无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于 malloc/free是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于malloc/free。

因此，C++需要一个能完成动态内存分配和初始化工作的运算符new，以及一个能完成清理与释放内存工作的运算符delete。注意：new/delete不是库函数。请看下面的例子。

```
1   #include <iostream>
2   using namespace std;
3
4   class Obj
5   {
6   public:
7       Obj(void)
8       {
9           cout << "Initialization" << endl;
```

```
10      }
11  ～Obj(void)
12      {
13          cout << "Destroy" << endl;
14      }
15  };
16
17  void UseMallocFree(void)
18  {
19      cout << "in UseMallocFree()..." << endl;
20      Obj *a = (Obj *)malloc(sizeof(Obj));
21      free(a);
22  }
23
24  void UseNewDelete(void)
25  {
26      cout << "in UseNewDelete()..." << endl;
27      Obj *a = new Obj;
28      delete a;
29  }
30
31  int main()
32  {
33      UseMallocFree();
34      UseNewDelete();
35
36      return 0;
```

37　}

在这个示例中，类 Obj 只有构造函数和析构函数的定义，这两个成员函数分别打印一句话。函数UseMallocFree()中调用malloc/free 申请和释放堆内存；函数UseNewDelete ()中调用new/delete申请和释放堆内存。可以看到函数UseMallocFree()执行时，类Obj的构造函数和析构函数都不会被调用；而函数UseNewDelete ()执行时，类Obj的构造函数和析构函数都会被调用。执行结果如下：

in UseMallocFree()...

in UseNewDelete()...

Initialization

Destroy

【答案】

对于非内部数据类型的对象而言，对象在消亡之前要自动执行析构函数。由于malloc/free是库函数而不是运算符，不在编译器控制权限之内，不能把执行构造函数和析构函数的任务强加于malloc/free，因此只有使用new/delete运算符。

面试题31 程序改错——指针的初始化

考点："野指针"必须初始化为NULL

出现频率：★★★

```
1    #include <stdio.h>
2    #include <malloc.h>
3
4    struct Tag_Node
5    {
6        struct Tag_Node* left;
7        struct Tag_Node* right;
8        int value;
9    };
10
11   typedef struct Tag_Node TNode;
12
13   TNode* root = NULL;
14
15   void append(int N);
16
17   int main()
18   {
19       append(63);
20       append(45);
21       append(32);
```

```
22      append(77);
23      append(96);
24      append(21);
25      append(17);
26      print();
27      return 0;
28  }
29
30  void append(int N)
31  {
32      TNode* NewNode = (TNode *)malloc(sizeof(TNode));
33      NewNode->value = N;
34
35      if(root == NULL)
36      {
37              root = NewNode;
38              return;
39      }
40      else
41      {
42              TNode* temp;
43              temp=root;
44
45              while((N >= temp->value && temp->left != NULL) ||
46                  (N < temp->value && temp->right != NULL))
47              {
48                  while(N >= temp->value && temp->left != NULL)
```

```
49                  temp = temp->left;
50                while(N < temp->value && temp->right != NULL)
51                  temp = temp->right;
52              }
53              if(N >= temp->value)
54                temp->left = NewNode;
55              else
56                temp->right = NewNode;
57              return;
58          }
59      }
```

【解析】

TNode是一个结构体类型，它有left和right两个成员指针，分别代表链接左、右两个元素，还有value成员表示元素节点的数据。在append函数中，它想把数据从左到右按降序排列。因此在第45、46、48和50行使用while循环来查找合适的位置。这里有一个问题，在这4行都采用temp的left或right与NULL进行判断，然而对堆中分配的内存只做了成员value的初始化（第33行），没有把left和right初始化为NULL，因此指针left和指针right与 NULL 进行的判断没有作用。结果是程序中会对野指针指向的地址进行赋值，从而导致程序崩溃。

改正后的代码如下：

```
1   #include <stdio.h>
2   #include <malloc.h>
3
4   struct Tag_Node
5   {
6       struct Tag_Node* left;
```

```
7       struct Tag_Node* right;
8       int value;
9     };
10    typedef struct Tag_Node TNode;
11
12    TNode* root = NULL;
13
14    void append(int N);
15    void print();
16
17    int main()
18    {
19        append(63);
20        append(45);
21        append(32);
22        append(77);
23        append(96);
24        append(21);
25        append(17);
26        printf("head: %d\n", root->value);
27        print();                    //打印链表所有元素
28    }
29
30    void append(int N)
31    {
32        TNode* NewNode = (TNode *)malloc(sizeof(TNode));
33        NewNode->value = N;
```

```
34      NewNode->left = NULL;      //初始化left
35      NewNode->right = NULL;      //初始化right
36
37      if(root == NULL)
38      {
39          root = NewNode;
40          return;
41      }
42      else
43      {
44          TNode* temp;
45          temp=root;
46
48                  (N < temp->value && temp->right != NULL))
47          while((N >= temp->value && temp->left != NULL) ||
49          {
50                  while(N >= temp->value && temp->left != NULL)
51                      temp = temp->left;
52                  while(N < temp->value && temp->right != NULL)
53                      temp = temp->right;
54          }
55          if(N >= temp->value)
56          {
57                  temp->left = NewNode;
58                  NewNode->right = temp;   //形成双向链表
59          }
60          else
```

```
61          {
62              temp->right = NewNode;
63              NewNode->left = temp;       //形成双向链表
64          }
65          return;
66      }
67  }
68
69  void print()
70  {
71      TNode* leftside = NULL;
72
73      if (root == NULL)
74      {
75          printf("There is not any element1");
76          return;
77      }
78
79      leftside = root->left;
80
81      while(1)
82      {
83          if (leftside->left == NULL)
84          {
85              break;
86          }
87          leftside = leftside->left;
```

```
88      }
89
90      while(leftside != NULL)
91      {
92          printf("%d ", leftside->value);
93          leftside = leftside->right;
94      }
95  }
```

如上面的程序所示，在第34、35行添加了成员指针left和right的初始化，这样就杜绝了野指针的产生。第58、63行的目的是为了使链表是双向链表。这样在遍历链表时就会比较方便。print函数是从左到右打印链表中所有元素value成员的。执行结果为：

```
head: 63
96 77 63 45 32 21 17
```

可以看到，root 节点是第一个插入到链表的，其数据值为 63。链表是按照从左到右降序排列的。

【答案】

没有对新增加的节点成员指针left和right做初始化，它们都是野指针，在随后与NULL比较时不起判断的作用。最终对野指针指向的内存块赋值导致程序崩溃。

面试题32 各种内存分配和释放的函数的联系和区别

考点：C语言的各种标准内存分配函数的使用

出现频率：★★★

【解析】

C语言的标准内存分配函数：malloc、calloc、realloc、free等。

malloc与calloc的区别为1块与n块的区别。

malloc 的调用形式为(类型*)malloc(size)：在内存的动态存储区中分配一块长度为“size”字节的连续区域，返回该区域的首地址，此时内存中的值没有初始化，是个随机数。

calloc 的调用形式为(类型*)calloc(n，size)：在内存的动态存储区中分配n 块长度为“size”字节的连续区域，返回首地址，此时内存中的值都被初始化为0。

realloc 的调用形式为(类型*)realloc(*ptr，size)：将 ptr 内存大小增大到 size，新增加的内存块没有初始化。

free 的调用形式为free(void*ptr)：释放ptr 所指向的一块内存空间。

C++中，new/delete函数可以调用类的构造函数和析构函数。

面试题33 程序找错——动态内存的传递

考点：动态内存的传递

出现频率：★★★★

```
1    #include <iostream>
2    using namespace std;
3
4    class Base
5    {
6    private:
7        char *name;
8    public:
9        Base(char * className)
10         {
11             name = new char[strlen(className)];
12             strcpy(name, className);
13         }
14     ～Base()
15         {
16             delete name;
17         }
18         char *copyName()
19         {
20             char newname[10] = "";
21             strcpy(newname, name);
```

```
22
23            return newname;
24        }
25        char *getName()
26        {
27            return name;
28        }
29    };
30    class Subclass : public Base
31    {
32    public:
33        Subclass(char * className) : Base(className)
34        {
35        }
36    };
37
38    int main()
39    {
40        Base *pBase = new Subclass("test");
41        printf("name: %s\n", pBase->getName());
42        printf("new name: %s\n", pBase->copyName());
43
44        return 0;
45    }
```

【解析】

这个程序有 Base 和 Subclass 两个类，其中 Subclass 是 Base 的子类。在本题里， Subclass只是被用来向Base的构造函数传递字符串参

数，以达到为Base的私有成员赋值的目的。

代码第11行，在Base的构造函数中用new分配了堆内存。其内存大小为传入的字符串长度，这里有个 bug。由于字符串是是以 0 作为结束符的，应该多分配一个字节存放0。

Base的成员函数copyName()中，返回其内数组的地址。由于数组处于栈中，当copyName调用结束后，栈就会被销毁。这里应返回堆内存地址。

【答案】

代码第11行改为：

name = new char[strlen(className) + 1];

代码第20行改为：

char *newname = new char[strlen(name) + 1];

修改后正确的输出为：

```
1    name: test
2    new name: test
```

面试题34 动态内存的传递

分析下面的代码。

考点：动态内存的传递

出现频率：★★★★

```
1    #include <iostream>
2    using namespace std;
3
4    void GetMemory(char *p, int num)
5    {
6        p = (char *)malloc(sizeof(char) *num);
7    };
8
9    int main(void)
10   {
11       char *str = NULL;
12
13       GetMemory(str, 10);
14       strcpy(str, "hello");
15
16       return 0;
17   }
```

【解析】

这里的GetMemory函数有问题。GetMemory函数体内的p实际上是main函数中的str变量在GetMemory函数栈中的一个备份，因为编译器总

是为函数的每个参数制作临时的变量。因此，虽然在代码第6行中p申请了堆内存，但是返回到main函数时，str还是NULL，并不指向那块堆内存。在代码第14行，调用strcpy时会导致程序崩溃。

实际上，GetMemory并不能做任何有用的事情。这里还要注意，由于从GetMemory函数返回时不能获得堆中内存的地址，那块堆内存就不能被继续引用，也就得不到释放，因此调用一次GetMemory函数就会产生num字节的内存泄漏。

可以采用3种方法来解决上面的动态内存不能传递的问题。

在C 语言中，可以通过采用指向指针的指针解决这个问题，可以把str的地址传给函数GetMemory。

在C++中，多了一种选择，就是传递str 指针的引用。

使用函数返回值来传递动态内存。

看下面的示例代码。

```
1   #include <iostream>
2   using namespace std;
3
4   void GetMemory(char *p, int num)
5   {
6       p = (char *)malloc(sizeof(char) *num);
7   };
8
9   void GetMemory2(char **p, int num)
10   {
11       *p = (char *)malloc(sizeof(char) *num);
12   };
13
14   void GetMemory3(char* &p, int num)
```

```
15  {
16      p = (char *)malloc(sizeof(char) *num);
17  };
18
19  char *GetMemory4(int num)
20  {
21      char *p = (char *)malloc(sizeof(char) *num);
22
23      return p;
24  }
25
26  int main(void)
27  {
28      char *str1 = NULL;
29      char *str2 = NULL;
30      char *str3 = NULL;
31      char *str4 = NULL;
32
33      //GetMemory(str1, 20);
34      GetMemory2(&str2, 20);
35      GetMemory3(str3, 20);
36      str4 = GetMemory4(20);
37
38      strcpy(str2, "GetMemory 2");
39      strcpy(str3, "GetMemory 3");
40      strcpy(str4, "GetMemory 4");
41
```

```
42      cout << "str1 == NULL? " << (str1 == NULL? "yes":"no") <<
endl;
43      cout << "str2:" << str2 << endl;
44      cout << "str3:" << str3 << endl;
45      cout << "str4:" << str4 << endl;
46
47      free(str2);
48      free(str3);
49      free(str4);
50      str2 = NULL;
51      str3 = NULL;
52      str4 = NULL;
53
54      return 0;
55   }
```

在上面的代码中，GetMemory2()函数采用二维指针作为参数传递；GetMemory3()函数采用指针的引用作为参数传递；GetMemory4()函数采用返回堆内存指针的方式。可以看到这3个函数都能起到相同的作用。

另外注意第47～52行，这里在主函数推出之前把指针str2、str3和str4指向的堆内存释放并把指针赋为NULL。每当决定不再使用堆内存时，应该把堆内存释放，并把指针赋为NULL，这样能避免内存泄漏以及产生野指针，是良好的编程习惯。

程序运行结果如下所示。

```
1   str1 == NULL? Yes
2   str2:GetMemory 2
3   str3:GetMemory 3
4   str3:GetMemory 4
```

【答案】

调用 strcpy(str, "hello")时程序崩溃。因为 GetMemory 不能传递动态内存，str 始终都是NULL。

面试题35 比较分析两个代码段的输出——动态内存的传递

考点：动态内存的传递

出现频率：★★★★★

程序1：

```
1    char *GetMemory()
2    {
3        char p[] = "hello world";
4        return p;
5    }
6
7    void Test(void)
8    {
9        char *str = NULL;
10         str = GetMemory();
11         printf(str);
12    }
```

程序2：

```
1    void GetMemory(char* p)
2    {
3        p=(char*)malloc(100);
4    }
5
6    void Test(void)
```

```
7   {
8       char *str=NULL;
9       GetMemory(str);
10      strcpy(str,"hello world");
11      printf(str);
12  }
```

【解析】

程序1的GetMemory()返回的是指向栈内存的指针，该指针的地址不是NULL，但是当栈退出后，内容不定，有可能会输出乱码。

程序2的GetMemory()没有返回值，这个函数不能传递动态内存。在Test函数中，str变量的值通过参数传值的方式赋给 GetMemory()的局部变量 p。但是 Test()中的 str 一直为NULL，所以第 10 行中的调用会使程序崩溃。此外，由于堆内存在 GetMemory()执行之后没有指针引用它，因此会产生内存泄露。

【答案】

程序1输出结果可能是乱码。

程序2有内存泄露，在Test函数调用strcpy时程序崩溃。

面试题36 程序查错——“野指针”用于变量值的互换

考点：“野指针”不能用于变量值的互换

出现频率：★★★

```
1   swap(int* p1, int* p2)
2   {
3       int *p;
4           *p = *p1;
5           *p1 = *p2;
6           *p2 = *p;
7   }
```

【解析】

在代码第3行，声明了一个指针p，由于没有对p初始化，p是个野指针，它可能指向系统区。因此在代码第 4 行，对 p 指向的内存区赋值非常危险，会导致程序运行时崩溃。程序应改为：

```
1   swap(int* p1, int* p2)
2   {
3       int p;
4       p = *p1;
5           *p1 = *p2;
6           *p2 = p;
7   }
```

【答案】

swap函数内的指针变量p没有初始化是野指针，野指针可能乱指一

气，导致程序运行时崩溃。

面试题37 内存的分配方式有几种

考点：静态存储区、栈、堆的内存分配

出现频率：★★★★★

【解析】

（1）从静态存储区域分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在，例如全局变量。

（2）在栈上创建。在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。处理器的指令集中有关于栈内存的分配运算，因此效率很高，但是分配的内存容量有限。

（3）从堆上分配，亦称动态内存分配。程序在运行的时候用malloc或new申请任意多少的内存，程序员自己负责在何时用 free 或 delete 释放内存。动态内存的生存期由我们决定，使用非常灵活，但问题也最多。

面试题38 什么是句柄

考点：对于Windows句柄的理解

出现频率：★★★

【解析】

句柄在 Windows 编程中是一个很重要的概念，在许多地方都扮演着重要的角色。在Windows环境中，句柄是用来标识项目的，这些项目包括：

模块（module）。

任务（task）。

实例（instance）。

文件（file）。

内存块（block of memory）。

菜单（menu）。

控制（control)。

字体（font）。

资源（resource），包括图标（icon）、光标（cursor）、字符串（string）等。

GDI对象（GDI object），包括位图（bitmap），画刷（brush）、元文件（metafile），调色板（palette）、画笔（pen）、区域（region），以及设备描述表（device context）。

Windows 是一个以虚拟内存为基础的操作系统。在这种系统环境下，Windows 内存管理器经常在内存中来回移动对象，以此来满足各种应用程序的内存需要。对象被移动意味着它的地址变化了。由于地址总是如此变化，所以Windows操作系统为各应用程序腾出一些内存储地

址，用来专门登记各应用对象在内存中的地址变化，而这地址（存储单元的位置）本身是不变的。Windows 内存管理器在移动对象在内存中的位置后，把对象新的地址告知这个句柄地址来保存。这样我们只需记住这个句柄地址就可以间接地知道对象具体在内存中的哪个位置。这个地址是在对象装载（Load）时由系统分配给的，当系统卸载时（Unload）又释放给系统。

因此，Windows 程序中并不是用物理地址来标识一个内存块、文件、任务或动态装入模块的，相反，WINDOWS API 给这些项目分配确定的句柄，并将句柄返回给应用程序，然后通过句柄来进行操作。

在Windows编程中会用到大量的句柄，比如HINSTANCE（实例句柄）、HBITMAP（位图句柄）、HDC（设备描述表句柄）、HICON（图标句柄）等。这当中还有一个通用的句柄，就是HANDLE，比如下面的语句：

```
1    HINSTANCE hInstance;
2    HANDLE hInstance;
```

句柄地址（稳定）→记载着对象在内存中的地址→对象在内存中的地址（不稳定）→实际对象。但是，必须注意的是，程序每次重新启动，系统不能保证分配给这个程序的句柄还是原来的那个句柄，而且绝大多数情况的确是不一样的。

面试题39 指针与句柄有什么区别

考点：对于Windows句柄的理解及其与一般指针的区别

出现频率：★★★★

【解析】

指针对应着一个数据在内存中的地址，得到了指针就可以自由地修改该数据。Windows 并不希望一般程序修改其内部数据结构，因为这样太不安全。所以 Windows给每个使用GlobalAlloc等函数声明的内存区域指定一个句柄，句柄是一种指向指针的指针。

句柄和指针都是地址，不同之处在于：

（1）句柄所指的可以是一个很复杂的结构，并且很有可能是与系统有关的。比如说线程的句柄，它指向的就是一个类或者结构，它和系统有很密切的关系。当一个线程由于不可预料的原因而终止时，系统就可以返回它所占用的资料，如 CPU、内存等。反过来想可以知道，这个句柄中的某一些项是与系统进行交互的。由于Windows系统是一个多任务的系统，它随时都可能要分配内存、回收内存、重组内存。

（2）指针也可以指向一个复杂的结构，但是通常是由用户定义的，所以必需的工作都要用户完成，特别是在删除的时候。

第4章 字符串

在C/C++中没有专门的字符串变量，通常用一个字符数组来存放一个字符串。字符串是以'\0'作为串的结束符。C/C++提供了丰富的字符串处理函数，下面列出了几个最常用的函数：

字符串输出函数puts；

字符串输入函数gets；

字符串连接函数strcat；

字符串复制函数strcpy；

测字符串长度函数strlen。

字符串是笔试以及面试的热门考点，通过字符串测试可以考查程序员的编程规范以及编程习惯。其中也包括了许多知识点，例如内存越界、指针与数组操作等等。许多公司在面试时会要求应试者写一段strcpy 复制字符串或字符串子串操作的程序。本章列举了一些与字符串相关的面试题及其解析，有些题要求较高的编程技巧。

面试题1 使用库函数将数字转换为字符串

考点：C语言库函数中数字转换为字符串的使用

出现频率：★★★

【解析】

C语言提供了几个标准库函数，可以将任意类型（整型、长整型、浮点型等）的数字转换为字符串。下面列举了各函数的方法及其说明。

itoa()：将整型值转换为字符串。

ltoa()：将长整型值转换为字符串。

ultoa()：将无符号长整型值转换为字符串。

gcvt()：将浮点型数转换为字符串，取四舍五入。

ecvt()：将双精度浮点型值转换为字符串，转换结果中不包含十进制小数点。

fcvt()：以指定位数为转换精度，其余同ecvt()。

还可以使用sprintf系列函数把数字转换成字符串，这种方式的速度比itoa()系列函数的速度慢。下面的程序演示了如何使用itoa()函数和gcvt()函数。

```
1   # include <stdio.h>
2   # include <stdlib.h>
3
4   int main ()
5   {
6       int num_int = 435;
7       double num_double = 435.10f;
8       char str_int[30];
```

9　　char str_double[30];

10

11　　itoa(num_int, str_int, 10);　　//把整数num_int转换成字符串str_int

12　　gcvt(num_double, 8, str_double);　//把浮点数num_double转换成字符串str_double

13

14　　printf("str_int: %s\n", str_int);

15　　printf("str_double: %s\n", str_double);

16

17　　return 0;

18　}

程序输出结果：

1　str_int: 435

2　str_double: 435.10001

代码第11 行中的参数10 表示按十进制类型进行转换，转换后的结果是"435"。如果按二进制类型进行转换，则结果为"1101110011"。

代码第12 行中的参数8 表示精确位数，这里得到的结果是"435.10001"。

【答案】

可以使用atoi系列的函数把数字转换成字符串。

面试题2 不使用库函数将整数转换为字符串

考点：对数字转换为字符串，相关ASCII码的理解

出现频率：★★★★

【解析】

如果不使用 atoi 或 sprintf 等库函数，我们可以通过把整数的各位上的数字加'0'转换成char 类型并存到字符数组中。但要注意，需要采用字符串逆序的方法。看下面的 C++程序示例。

```
1   #include <iostream>
2   using namespace std;
3
4   void int2str(int n, char *str)
5   {
6       char buf[10] = "";
7       int i = 0;
8       int len = 0;
9       int temp = n < 0 ? -n: n;          // temp为n的绝对值
10
11       if (str == NULL)
12       {
13           return;
14       }
15       while(temp)
16       {
17           buf[i++] = (temp % 10) + '0';    //把temp的每一位上的数存
```

入buf

```
18          temp = temp / 10;
19      }
20
21      len = n < 0 ? ++i: i;           //如果n是负数，则多需要一位来
存储负号
22      str[i] = 0;                     //末尾是结束符0
23      while(1)
24      {
25          i--;
26          if (buf[len-i-1] ==0)
27          {
28                  break;
29          }
30          str[i] = buf[len-i-1];      //把buf数组里的字符拷到字符串
31      }
32      if (i == 0 )
33      {
34              str[i] = '-';           //如果是负数，添加一个负号
35      }
36  }
37
38  int main()
39  {
40      int nNum;
41      char p[10];
42
```

```
43        cout << "Please input an integer:";
44        cin >> nNum;
45        cout << "output: " ;
46        int2str(nNum, p);                //整型数转换成字符串
47        cout<< p << endl;
48
49        return 0;
50    }
```

这个程序中的int2str函数完成了int类型到字符串的转换。在main函数的第46行对int2str函数做了测试。程序的执行结果：

```
1    Please input an integer: 1234
2    Output: 1234
```

如果输入的是个负数，例如

```
1    Please input an integer: -1234
2    Output: -1234
```

接下来对int2str函数的实现进行分析。

代码第9 行，把传入参数n 的绝对值赋给temp，以后在计算各个位的整数时就用temp了，这样保证在负数情况下取余不会出现问题。

代码第11～14 行判断str 的有效性，str 应不为NULL。

代码第 15～19 行的 while 循环中，将 n 的各个位存放到局部数组 buf 中，存放的顺序与整数顺序相反。例如 n 为整数 123456，while 循环结束后 buf 应为"654321"。

代码第21 行计算实际转换后的字符串长度len，如果是负数，长度应该再加1。

代码第22～31行把数组buf中的非0元素逆向复制到参数str指向的内存中，如果n是负数，则保留str指向的第一个内存以存放负号'-'。

最后在第34 行里，如果是负数，添加负号'-'到str 开头。

面试题3 使用库函数将字符串转换为数字

考点：C语言库函数中字符串转换为数字的使用

出现频率：★★★★

【解析】

与上题数字转换为字符串类似，C/C++提供了几个标准库函数，可以将字符串转换为任意类型（整型、长整型、浮点型等）的数字。下面列举了各函数的方法及其说明。

atof()：将字符串转换为双精度浮点型值。

atoi()：将字符串转换为整型值。

atol()：将字符串转换为长整型值。

strtod()：将字符串转换为双精度浮点型值，并报告不能被转换的所有剩余数字。

strtol()：将字符串转换为长整型值，并报告不能被转换的所有剩余数字。

strtoul()：将字符串转换为无符号长整型值，并报告不能被转换的所有剩余数字。

下面的程序演示了如何使用atoi ()函数和atof ()函数。

```
1    # include <stdio.h>
2    # include <stdlib.h>
3
4    int main ()
5    {
6        int num_int;
7        double num_double;
```

```
8       char str_int[30] = "435";           //将要被转换为整型值的字符串
9       char str_double[30] = "436.55";   //将要被转换为浮点型值的字符串
10
11        num_int = atoi(str_int);          //转换为整型值
12        num_double = atof(str_double);       //转换为浮点型值
13
14        printf("num_int: %d\n", num_int);
15        printf("num_double: %lf\n", num_double);
16
17        return 0;
18    }
```

输出结果：

```
1    num_int: 435
2    num_double: 436.550000
```

面试题4 不使用库函数将字符串转换为数字

考点：对字符串转换为数字，相关ASCII码的理解

出现频率：★★★★

【解析】

程序代码如下。

```
1    #include <iostream>
2    using namespace std;
3
4    int str2int(const char *str)
5    {
6            int temp = 0;
7            const char *ptr = str;                    //ptr保存str字符串开头
8
9            if (*str == '-' || *str == '+')           //如果第一个字符是正负
号，
10       {                                        //则移到下一个字符
11               str++;
12       }
13       while(*str != 0)
14       {
15               if ((*str < '0') || (*str > '9'))    //如果当前字符不是数字，
16               {                                 //则退出循环
17                   break;
18               }
```

19　　　　temp = temp * 10 + (*str - '0');　//如果当前字符是数字，则计算数值

20　　　　str++;　　　　　　　　//移到下一个字符

21　　}

22　　if (*ptr == '-')　　　　　　//如果字符串以'-'开头，则转换成其相反数

23　　{

24　　　　temp = -temp;

25　　}

26

27　　return temp;

28　}

29

30　int main()

31　{

32　　int n = 0;

33　　char p[10] = "";

34

35　　cin.getline(p, 20);　//从终端获取一个字符串

36　　n = str2int(p);　　//把字符串转换成整型数

37

38　　cout << n << endl;

39

40　　return 0;

41　}

程序执行结果：

1　输入：1234

2　输出：1234
3　输入：-1234
4　输出：-1234
5　输入：+1234
6　输出：1234

上面程序中的str2int函数用于将字符串转换成整数。这个函数的转换过程与面试题2中的int2str函数相比更加简单。它没有逆序的需要，只需要做一次while循环（代码第13行）就能把数值大小计算出来。如果最后是负数，加一个负号。

面试题5 编程实现strcpy函数

考点：字符串复制的实现

出现频率：★★★★

已知strcpy函数的原型是：

char * strcpy(char * strDest,const char * strSrc);

（1）不调用库函数，实现strcpy函数。

（2）解释为什么要返回char *。

【解析】

代码如下。

```
1    #include <stdio.h>
2
3    char * strcpy(char * strDest, const char * strSrc)    //实现strSrc到
strDest的复制
4    {
5            if ((strDest == NULL) || (strSrc == NULL))    //判断参数
strDest和strSrc的有效性
6            {
7                return NULL;
8            }
9            char *strDestCopy = strDest;              //保存目标字符串的首
地址
10         while ((*strDest++ = *strSrc++)!='\0'); //把strSrc字符串的内容
复制到strDest下
11
```

```
12      return strDestCopy;
13  }
14
15  int getStrLen(const char *strSrc)              //实现获取strSrc字符串
的长度
16  {
17      int len = 0; //保存长度
18      while(*strSrc++ != '\0')              //循环直到遇见结束符'\0'为止
19      {
20              len++;
21      }
22
23      return len;
24  };
25
26  int main()
27  {
28      char strSrc[] = "Hello World!";        //要被复制的源字符串
29      char strDest[20];                      //要复制到的目的字符数组
30      int len = 0;                           //保存目的字符数组中字符串的长
度
31
32      len = getStrLen(strcpy(strDest, strSrc)); //链式表达式，先复制
后计算长度
33      printf("strDest: %s\n", strDest);
34      printf("Length of strDest: %d\n", len);
35
```

```
36        return 0;
37    }
```

（1）strcpy函数的实现说明：

代码第 5～7 行判断传入的参数 strDest 和 strSrc 是否为 NULL，如果是则返回NULL。

代码第9 行把strDest 的值保存到strDestCopy。

代码第10 行对strSrc 和strDest 两个指针做循环移动，并不断复制strSrc 内存的值到strDest内存中。

由于第2 步中保存了strDest 的值，因此这里只需返回strDestCopy，那么函数调用完后返回的就是strDest的值。

（2）为什么strcpy 函数要返回char *类型呢？这是为了能使用链式表达式。由于在strcpy中使用了char *返回类型，因此在代码第32 行中可以通过这种链式表达式来同时做两个操作。首先调用strcpy，使得strDest复制了strSrc指向的内存数据，然后调用getStrLen函数获取 strDest字符串的长度。这样不仅调用方便，而且程序结构简洁明了。程序的输出结果如下。

```
1    strDest: Hello World!
2    Length of strDest: 12
```

面试题6 编程实现memcpy函数

考点：内存复制的实现

出现频率：★★★★

【答案】

程序代码如下所示。

```
1    #include <stdio.h>
2    #include <assert.h>
3
4    void *memcpy2(void *memTo, const void *memFrom, size_t size)
5    {
6            assert((memTo != NULL) && (memFrom != NULL));
//memTo和memFrom必须有效
7            char *tempFrom = (char *)memFrom;          //保存memFrom
首地址
8            char *tempTo = (char *)memTo;              //保存memTo首地
址
9
10       while(size -- > 0)          //循环size次，复制memFrom的值到
memTo中
11                *tempTo++ = *tempFrom++ ;
12
13       return memTo;
14    }
15
```

```
16  int main()
17  {
18      char strSrc[] = "Hello World!";   //将被复制的字符数组
19      char strDest[20];                 //目的字符数组
20
21      memcpy2(strDest, strSrc, 4);      //复制strSrc的前4个字符到strDest中
22      strDest[4] = '\0';                //把strDest的第5个元素赋为结束符'\0'
23      printf("strDest: %s\n", strDest);
24
25      return 0;
26  }
```

memcpy的实现如下：

与strcpy不同，memcpy以参数size决定复制多少个字符（strcpy是遇见结束符'\0'结束）。由于在主程序中只复制了strSrc的前4个字符（代码第22行），程序输出如下。

```
1   strDest: Hell
```

面试题7 strcpy与memcpy的区别

考点：字符串复制与内存复制之间的区别

出现频率：★★★★

【解析】

主要有下面几方面的区别。

复制的内容不同。strcpy只能复制字符串，而memcpy可以复制任意内容，例如字符数组、整型、结构体、类等。

复制的方法不同。strcpy 不需要指定长度，它是遇到字符串结束符'\0'而结束的。memcpy则是根据其第三个参数决定复制的长度。

用途不同。通常在复制字符串时用strcpy；而若需要复制其他类型数据，则一般用memcpy。

面试题8 改错——数组越界

考点：数组越界出现的问题

出现频率：★★★★

试题1

```
1    void test1()
2    {
3        char string[10];
4        char* str1 = "0123456789";
5        strcpy(string, str1);
6    }
```

试题2

```
1    void test2()
2    {
3        char string[10], str1[10];
4        int i;
5        for(i=0; i<10; i++)
6        {
7                str1[i] = 'a';
8        }
9        strcpy(string, str1);
10   }
```

试题3

```
1    void test3(char*str1)
2    {
```

```
3       char string[10];
4       if(strlen(str1) <= 10)
5       {
6           strcpy(string, str1);
7       }
8   }
```

【解析】

这3道题都有数组越界的问题。

试题1 中，string 是一个含有10 个元素的字符数组，str1 指向的字符串长度为10，在进行strcpy调用时，会将str1的结束符也复制到string数组里，也就是说会复制的字符数为 11，这样就会导致 string 出现数组越界。此时程序不一定会因此而崩溃，但这是一个潜在的危险。解决办法：将string的元素个数定义为11个。

试题2 中，str1 和 string 都是一个含有 10 个元素的字符数组，并且str1 的元素全部被赋为字符'a'，然后再调用strcpy。这里会出现以下两个问题：一个是str1表示的字符数组没有以'\0'结束，在随后调用 strcpy 时无法判断什么时候复制结束；另一个是string的数组长度不够，与试题1出现类似数组越界。解决办法：将string和str1的元素个数都定义为11个，并在调用strcpy之前加入一条语句把str1[10]赋为'\0'。

试题3 中，其中的if 语句用的是小于等于“<=”比较，这里如果str1的长度等于10，也会出现试题1中数组越界的情况。解决办法：把“<=”换成“<”。

【答案】

3道题都有数组越界的问题。改正后的程序如下。

```
1   #include <stdio.h>
2   #include <string.h>
3
```

```
4   void test1()
5   {
6       char string[11];        //字符数组长度为11，多分配一个
7       char* str1 = "0123456789";
8       strcpy(string, str1);
9   }
10
11  void test2()
12  {
13      char string[11], str1[11];    //字符数组长度都为11，均多分配
一个
14      int i;
15      for(i=0; i<10; i++)
16      {
17          str1[i] = 'a';
18      }
19      str1[10] = '\0';            //初始化str1为空字符串
20      strcpy(string, str1);
21  }
22
23  void test3(char*str1)
24  {
25      char string[10];
26      if(strlen(str1) < 10)           //不能用<=
27      {
28          strcpy(string, str1);
29      }
```

```
30  }
31
32  int main()
33  {
34      test1();
35      test2();
36      test3("1234");
37
38      return 0;
39  }
```

面试题9 分析程序——数组越界

考点：不当的循环操作导致数组越界

出现频率：★★★

下面这个程序执行后会出现什么错误或者效果。

```
1  #define MAX 255
2  int main()
3  {
4     unsigned char A[MAX], i;
5
6     for (i = 0; i <= MAX; i++)
7           A[i] = i;
8  }
```

【解析】

代码第6行的for循环中用的是“<=”，当i=MAX时发生数组越界。注意：这个程序很容易使人误认为只有数组越界的问题，但只要再细心些就能发现，i是无符号的char类型，它的范围是0～255，所以i<=MAX一直都是真，这样会导致无限循环。可以把“i<=MAX”改为i<MAX，这样既避免了无限循环，又避免了数组越界。

【答案】

i<=MAX导致数组越界以及无限循环，应改为i<MAX。

面试题10 分析程序——打印操作可能产生数组越界

考点：打印操作时可能产生的数组越界问题

出现频率：★★★

下面这个程序的打印结果是什么？

```
1    #include <stdio.h>;
2
3    int main()
4    {
5        int a[5]={0, 1, 2, 3, 4}, *p;
6        p = a;
7        printf("%d\n",*(p + 4*sizeof(int)));
8
9        return 0;
10   }
```

【答案】

这个程序存在着越界的问题。

代码第6行，p指向a的第一个元素，所以p+4指向a的最后一个元素，即4，p + 4 * sizeof(int)即p+16，此时指向的是数组a的第17个元素，显然已经越界了，因此打印的是个随机数。

面试题11 编程实现计算字符串的长度

考点：strcpy库函数的实现细节

出现频率：★★★★★

【解析】

这个题目非常简单。我们知道字符串是以'\0'作为结束符的，所以只需要做一次遍历就可以了。但是需要注意的是，要尽量把程序写得简单且效率高。看下面的示例代码：

```
1   #include <stdio.h>
2   #include <assert.h>
3
4   int strlen1(const char* src)
5   {
6          assert( NULL != src);        //src必须有效
7          int len = 0;                 //保存src的长度
8          while(*src++ != '\0')        //遇到结束符'\0'时退出循环
9              len++;                   //每循环一次，len加1
10      return len;
11    }
12
13    int strlen2(const char* src)
14    {
15      assert( NULL != src);        //src必须有效
16      const char *temp = src;        //保存src首地址
17      while(*src++ != '\0');        //遇到结束符'\0'时退出循环
```

```
18      return (src-temp-1);          //返回尾部指针与头部指针之差，即长度
19   }
20
21   int main()
22   {
23      char p[] = "Hello World!";
24      printf("strlen1 len: %d\n", strlen1(p));    //打印方法1得到的结果
25      printf("strlen2 len: %d\n", strlen2(p));    //打印方法2得到的结果
26
27      return 0;
28   }
```

strlen1和strlen2这两个函数都可以用来计算字符串长度。下面来比较它们的区别：

strlen1用一个局部变量len在遍历的时候做自增，然后返回len。因此，每当while循环一次，就需要执行两次自增操作。

strlen2用一个局部变量temp记录src遍历前的位置。while循环一次只需要一次自增操作。最后返回指针之间的位置差。

显然，strlen2比strlen1的效率要高，尤其是在字符串较长的时候。下面是程序的输出结果。

```
1   strlen1 len: 12
2   strlen2 len: 12
```

面试题12 编程实现字符串中子串的查找

考点：strstr库函数的实现细节

出现频率：★★★★★

请写一个函数，实现 strstr ，即从一个字符串中，查找另一个字符串的位置，如strstr("12345", "34")返回值为2，在2 号位置找到字符串34。

【解析】

程序如下所示。

```
1    #include <stdio.h>
2    #include <assert.h>
3
4    const char *strstr(const char* src, const char* sub)
5    {
6            const char *bp;
7            const char *sp;
8
9            if (src == NULL || NULL == sub) //判断src与sub的有效性
10       {
11               return src;
12       }
13       while (*src) //遍历src字符串
14       {
15               bp = src;                  //用于src的遍历
16               sp = sub;                  //用于sub的遍历
```

```
17          do
18          {                       //遍历sub字符串
19              if (!*sp)           //如果到了sub字符串结束符位置
20                  return src;   //表示找到了sub字符串，退出
21          } while (*bp++ == *sp++);
22          src += 1;
23      }
24
25      return NULL;
26  }
27  int main()
28  {
29      char p[] = "12345";
30      char q[] = "34";
31
32      char *r = strstr(p, q);
33      printf("r: %s\n", r);
34
35      return 0;
36  }
```

main函数中的测试结果为：

```
1   r: 345
```

可以看出，第32行调用strstr结束之后，r指向了数组p的第3个元素。这里strstr函数的方法是循环取src的子串与sub比较。以本题中的"12345"和"34"为例，比较步骤如下。

（1）"12345"和"34"比较，不满足匹配。

（2）"2345"和"34"比较，不满足匹配。

（3）"345"和"34"比较，满足匹配。

面试题13 编程实现字符串中各单词的翻转

考点：字符串相关的综合编程能力

出现频率：★★★

编写函数，将"I am from Shanghai"倒置为"Shanghai from am I"，即句子中的单词位置倒置，而不改变单词内部的结构。

【解析】

第一种方法的代码如下。

```
1   #include <iostream>
2   using namespace std;
3
4   void RevStr(char *src)
5   {
6       char *start = src, *end = src, *ptr = src;
7
8       while(*ptr++ != '\0')                //遍历字符串
9       {
10              if(*ptr == ' ' || *ptr == '\0')    //找到一个单词
11              {
12                  end = ptr - 1;              //end指向单词末尾
13                  while(start < end)
14                  swap(*start++, *end--);    //把单词的字母逆置
15
16                  start = end = ptr+1;       //指向下一个单词开头
17              }
```

```
18        }
19        start = src, end = ptr-2;            //start指向字符串开头，end指向字符串末尾
20        while(start < end)
21        {
22            swap(*start++, *end--);          //把整个字符串逆置
23        }
24    }
25
26    int main()
27    {
28        char src[] = "I am from Shanghai";
29        cout << src << "\n";
30        RevStr(src);
31        cout << src << "\n";
32
33        return 0;
34    }
```

程序输出结果：

```
1    I am from Shanghai
2    Shanghai From am I
```

第一种方法RevStr函数有两个转换步骤：代码第8～18行将src中所有的单词进行翻转，其结果是src 的内容变为"I ma morf iahgnahS"，然后代码第20～23 行再进行全局翻转。

第二种方法的代码如下。

```
1    #include <iostream>
2    using namespace std;
```

```
3
4   void RevStr(char *src)
5   {
6       char *start = src, *end = src, *ptr = src;
7
8       while(*ptr++ != '\0');
9       end = ptr-2;                          //找到字符串末尾
10        while(start < end)
11        {
12            swap(*start++, *end--);           //逆置整个字符串
13        }
14
15        start = src, end = ptr-2;
16        ptr = start;
17        while(*ptr++ != '\0')
18        {
19            if(*ptr == ' ' || *ptr == '\0')    //找到单词
20            {
21                    end = ptr - 1;                //end指向单词末尾
22                    while(start < end)
23                    swap(*start++, *end--);       //逆置单词
24
25                    start = end = ptr+1;          //指向下一个单词开头
26            }
27        }
28    }
29
```

```
30    int main()
31    {
32        char src[] = "I am from Shanghai";
33        cout << src << "\n";
34        RevStr(src);
35        cout << src << "\n";
36
37        return 0;
38    }
```

程序输出结果：

```
1    I am from Shanghai
2    Shanghai from am I
```

第二种方法RevStr函数转换步骤与第一种方法相反：代码第10～11行将src进行全局翻转。其结果是src 的内容变为"iahgnahS morf ma I"，然后代码第17～27行再把所有的单词进行翻转。

从上面的代码分析可以看出，两种方法都是采用两个步骤，即字符串全局翻转和各个单词局部翻转，只是步骤的顺序不同，得到的结果都是一致的。

面试题14 编程判断字符串是否为回文

考点：字符串相关的综合编程能力

出现频率：★★★★

判断一个字符串是不是回文，例如单词“level”是回文。

【解析】

根据题目要求，我们可以从一个字符串的两端进行遍历比较。例如，对于"level"字符串，我们可以进行下面的步骤。

（1）计算需要比较的次数。由于"level"字符串长度为5，是奇数，因此比较两次。

（2）第一次比较：看"level"的第一个字符与最后一个字符是否相等，若相等，则进行第二次比较。

（3）第二次比较：看"level"的第二个字符与倒数第二个字符是否相等，若相等，则是回文。

如果在上面的比较过程中有一个不相等，则字符串不是回文。根据上面的思路，我们可以写出如下的程序代码。

```
1   #include <iostream>
2   using namespace std;
3
4   int IsRevStr(char *str)
5   {
6           int i, len;
7           int found = 1;                    //1表示是回文字符串，0表示不
是
8
```

```
9          if(str == NULL)                //判断str的有效性
10       {
11             return -1;
12       }
13       len = strlen(str);               //获得字符串长度
14       for(i=0; i<len/2; i++)
15       {
16             if(*(str+i) != *(str+len-i-1))   //遍历中如果发现相应的头
尾字符不等，
17             {                        //则字符串不是回文
18                 found=0;
19                 break;
20             }
21       }
22       return found;
23   }
24
25   int main()
26   {
27       char str1[10] = "1234321";         //回文字符串
28       char str2[10] = "1234221";         //非回文字符串
29
30       int test1 = IsRevStr(str1);      //测试str1是不是回文
31       int test2 = IsRevStr(str2);      //测试str2是不是回文
32
33       cout << "str1 is " << (test1 == 1 ? "": "not ")
34                             << "reverse string." << endl;
```

```
35        cout << "str2 is " << (test2 == 1 ? "": "not ")
36                                  << "reverse string." << endl;
37
38        return 0;
39    }
```

这个程序中的 IsRevStr()函数用于判断字符串是否是回文字符串，如果是，则返回 1，否则返回0。输出结果：

```
1    str1 is reverse string.
2    str2 is not reverse string.
```

面试题15 编程实现stcmp库函数

考点：库函数strcmp的实现细节

出现频率：★★★★★

【解析】

此题实际上就是做一个C/C++库函数中的strcmp的实现。对于两个字符串str1和str2，若相等，则返回0，若str1大于str2，则返回1，若str1小于str2，则返回−1。

程序代码如下。

```
1    #include <iostream>
2    using namespace std;
3
4    int mystrcmp(const char *src, const char *dst)
5    {
6        int ret = 0 ;
7        while( !(ret = *(unsigned char *)src - *(unsigned char *)dst) &&
*dst)
8        {                                //循环比较两个字符是否相等
9                ++src;                   //如果不等或者到了dst字符串末尾，
则
10           ++dst;                       //退出循环
11       }
12       if ( ret < 0 )                   //ret保存着字符比较的结果
13               ret = -1 ;
14       else if ( ret > 0 )
```

```
15              ret = 1 ;
16          return( ret );
17      }
18
19      int main()
20      {
21          char str[10] = "1234567";
22          char str1[10] = "1234567";          //str1 == str
23          char str2[10] = "12345678";        //str2 > str
24          char str3[10] = "1234566";          //str3 < str
25
26          int test1 = mystrcmp(str, str1);    //测试str与str1比较
27          int test2 = mystrcmp(str, str2);    //测试str与str2比较
28          int test3 = mystrcmp(str, str3);    //测试str与str3比较
29
30          cout << "test1 = " << test1 << endl;
31          cout << "test2 = " << test2 << endl;
32          cout << "test3 = " << test3 << endl;
33
34          return 0;
35      }
```

mystrcmp()函数对 src 和 dst 两个字符串同时进行了一次遍历，当发现它们存在不同值时停止循环，最后根据它们的最后一个字符的大小，返回相应的结果。程序输出结果：

```
1   test1 = 0
2   test2 = -1
3   test3 = 1
```

面试题16 编程查找两个字符串的最大公共子串

考点：字符串相关的综合编程能力

出现频率：★★★

【解析】

对于两个字符串A和B，如果A="aocdfe"，B="pmcdfa"，则输出"cdf"。

```
1    #include<stdio.h>
2    #include<stdlib.h>
3    #include<string.h>
4
5    char *commonstring(char *str1, char *str2)
6    {
7        int i, j;
8        char *shortstr, *longstr;
9        char *substr;
10
11         if (NULL == str1 || NULL == str2)    //判断str1与str2的有效性
12         {
13             return NULL;
14         }
15
16         if (strlen(str1) <= strlen(str2)) //shortstr和longstr分别指向较短
和较长的字符串
```

17　　{

18　　　　shortstr = str1;

19　　　　longstr = str2;

20　　} else

21　　{

22　　　　shortstr = str2;

23　　　　longstr = str1;

24　　}

25

26　　if(strstr(longstr, shortstr) != NULL)　//如果在长的字符串中能寻找短的字符串，

27　　{　　　　　　　　　　//返回短字符串

28　　　　return shortstr;

29　　}

30

31　　substr = (char *)malloc(sizeof(char) * (strlen(shortstr) + 1)); //申请堆内存存放返回结果

32

33　　for(i=strlen(shortstr)-1; i>0; i--)

34　　{

35　　　　for(j=0; j<=strlen(shortstr)-i; j++)

36　　　　{

37　　　　　memcpy(substr, &shortstr[j], i);　//将短字符串的一部分复制到substr，

38　　　　　substr[i] = '\0';　　　　//其长度逐渐减小

39　　　　　if(strstr(longstr,substr)!=NULL)//如果在longstr中能找到substr，则返回substr

```
40                    return substr;
41              }
42          }
43
44          return NULL;
45      }
46
47      int main()
48      {
49          char *str1 = (char *)malloc(256);       //分配堆内存存放字符串
str1和str2
50          char *str2 = (char *)malloc(256);
51          char *common=NULL;
52
53          gets(str1);                             //从终端输入str1和str2
54          gets(str2);
55
56          common = commonstring(str2, str1);       //取最大的相同子串
57
58          printf("the longest common string is: %s\n", common);
59
60          return 0;
61      }
```

为了方便，我们可以利用库函数 strstr 中一个字符串寻找子串。这个程序的步骤如下。

（1）代码第11～14行，检查参数str1和str2的有效性。

（2）计算两个字符串的长短，这样在调用strstr时就会比较方便。

（3）调用strstr直接进行两个字符串的整串比较，如果不为NULL，说明短串被长串所包含，直接返回短串即可，否则进行下一步。

（4）申请一块大小为短串长度加1的堆内存，这块内存用于保存最大公共子串。

（5）循环取短串的子串放入堆内存，调用strstr函数检查长串中是否包含这个子串，如果有，则返回堆内存。注意短串的长度是不断减小的。

下面是程序执行结果。

```
1    aocdfe（输入）
2    pmcdfa（输入）
3    the longest common string is: cdf
```

面试题17 不使用printf，将十进制数以二进制和十六进制的形式输出

考点：用字符串表示十进制数

出现频率：★★★

【解析】

如果不能使用 printf 系列库函数，我们可以通过位运算得到这个十进制数的二进制和十六进制形式的字符串，再将字符串打印。源代码如下。

```
1    #include<stdio.h>
2    #include<stdlib.h>
3    #include<string.h>
4
5    char *get2String(long num)          //得到二进制形式的字符串
6    {
7            int i=0;
8            char* buffer;
9            char* temp;
10
11        buffer = (char*)malloc(33);
12        temp = buffer; //temp
13        for (i=0; i < 32; i++)
14        {                          //给数组的32个元素赋'0'或'1'
15              temp[i] = num & (1 << (31 - i));
16              temp[i] = temp[i] >> (31 - i);
```

```
17              temp[i] = (temp[i] == 0) ? '0': '1';
18          }
19          buffer[32] = '\0';           //字符串结束符
20
21          return buffer;
22      }
23
24      char *get16String(long num)           //得到十六进制形式的字符串
25      {
26          int i=0;
27          char* buffer = (char*)malloc(11);
28          char* temp;
29
30          buffer[0] = '0'; //”0x”开头
31          buffer[1] = 'x';
32          buffer[10] = '\0';           //字符串结束符
33          temp = buffer + 2;
34
35          for (i=0; i < 8; i++)
36          {                            //给数组的8个元素赋值
37              temp[i] = (char)(num<<4 * i>>28);
38              temp[i] = temp[i] >= 0 ? temp[i] : temp[i] + 16;
39              temp[i] = temp[i] < 10 ? temp[i] + 48 : temp[i] + 55;
40          }
41          return buffer;
42      }
43
```

```
44  int main()
45  {
46      char *p = NULL;
47      char *q = NULL;
48      int num = 0;
49
50      printf("input num: ");
51      scanf("%d", &num);          //输入整数
52
53      p = get16String(num);          //得到十六进制形式的字符串
54      q = get2String(num);          //得到二进制形式的字符串
55
56      printf("%s\n", p);
57      printf("%s\n", q);
58
59      return 0;
60
}
```

这个程序中，get2String和get16String分别用于得到二进制字符串和十六进制字符串。

long型的整数是4个字节，32位，每一位用0或1表示。get2String函数申请了33个字节（包括'\0'）的堆内存存放结果，get16String函数申请了11个字节（包括'0'、'x'和'\0'）。然后把每个位的值赋给堆内存的对应位置。程序执行结果：

```
1   input num: 456789（输入）
2   0x0006F855
3   00000000000000000000000001010101
```

面试题18 编程实现转换字符串、插入字符的个数

考点：字符串相关的综合编程能力

出现频率：★★★★

【解析】

根据题意，需要在字符串中插入字符统计的个数。例如字符串aaab，插入字符个数后变成aaa3b1。

源程序如下。

```
1    #include <stdlib.h>
2    #include <stdio.h>
3    #include <string.h>
4
5    #define MAXCOUNT 2*100
6
7    char *transformation(char *str)
8    {
9            int len=strlen(str);
10       char *buf=new char[len+1];
11
12       char *p=str;
13       char *q=p+1;
14       int count=1;
15       while(*q)
16       {
```

```
17              if(*p==*q)
18              {
19                  count++;
20                  p++;
21                  q++;
22              }
23              else
24              {
25                  itoa(count,buf,10);
26                  int nbits=strlen(buf);
27                  strcat(buf,q);
28                      *q=0;
29                  strcat(str,buf);
30                  q+=nbits;
31                  p=q;
32                  q=p+1;
33                  count=1;
34              }
35          }
36
37          itoa(count,buf,10);
38          strcat(str,buf);
39
40          delete []buf;
41          buf=NULL;
42          return str;
43      }
```

```
44
45  int main()
46  {
47      char str[MAXCOUNT];
48
49      printf("please input a string:");
50      scanf("%s",&str);
51      printf("before transformation: %s\n",str);
52      char *pstr=transformation(str);
53      printf("after    transformation: %s\n",pstr);
54
55      return 0;
56  }
```

这个程序的transformation()函数用来转换字符串。我们以字符串aaab 为例来说明其执行过程。为了计算方便，首先申请了5个字节的堆内存buf（aaab长度为4，加一个结束符为5个）来存放字符串数字相关的信息。初始计数为1，然后进行下面的步骤：

遍历aaab，直到找到不同的字符，然后在buf中保存3b，把str变为aaa（字符'b'位置内存设为0）。然后执行strcat(str, buf)，此时str 变为aaa3b，计数设为1。如果到字符串末尾（碰到结束符'\0'），则退出循环，否则继续进行以上的步骤。

如果退出循环，则将最后一个字符个数存入buf（这里为b的个数1），此时str中为aaa3b并且调用strcat(str, buf)，结果str 变为aaa3b1。

释放buf堆内存并返回str。程序执行结果为：

```
please input a string:aaab
before transformation: aaab
after transformation: aaa3b1
```

面试题19 字符串编码例题

考点：字符串相关的综合编程能力

出现频率：★★

Give an implementation of encoding a string which contains less than 20 chars.There threerules:

1.replace the alphabetical char in the string with the fourth char behind it, for example, a -> e ， A -> E，X -> B, y -> c, z -> d

2.if the char is not a alphabetical char, ignore it.

3.reverse the string updated

【解析】

下面是原题的中文翻译。

对一个长度小于20的字符串进行编码，遵循3个规则：

（1）把字符串中的字母替换成它的第4个字母。例如，a->e，A -> E，X -> b，y ->c，z ->d。

（2）如果字符不是字母，忽略替换。

（3）翻转整个字符串。

整个过程可以分为两部分：替换字符和翻转字符串，其中替换字符还包括一个检查的操作，即检查是否是在字母表中的字符，也就是英文的26个字符。程序示例：

```
1    #include <iostream>
2    using namespace std;
3
4    char LowerCaseAlphabets[] =
5                  {'a','b','c','d','e','f','g','h',
```

```
6                    'i','j','k','l','m','n','o','p',
7                    'q','r','s','t','u','v','w','x','y','z'};
8   char UpperCaseAlphabets[]=
9                    {'A','B','c','D','E','F','G','H',
10                     'I','J','K','L','M','N','O','P',
11                     'Q','R','S','T','U','V','W','X','Y','Z'};
12
13   char GetFourthChar(char chrSource,char alphabets[])
14   {
15       for(int i=0;i<26;i++)
16       {
17               if(alphabets[i]==chrSource)
18               {
19                   int index = (i+4) % 26;
20                   return alphabets[index];
21               }
22       }
23       return '\0';
24   };
25
26   void ReplaceChars(char chars[], int len)
27   {
28       for(int i=0; i<len; i++)
29       {
30               if('a' <= chars[i] && chars[i] <= 'z')
31               {
32                   chars[i]=GetFourthChar(chars[i], LowerCaseAlphabets);
```

```
33              }
34              else if('A' <= chars[i] && chars[i] <= 'Z')
35              {
36                  chars[i]=GetFourthChar(chars[i], UpperCaseAlphabets);
37              }
38      }
39  };
40
41  void ReverseString(char str[],int len)
42  {
43      int begin=0, end=len-1;
44      if(str[end] == '\0')
45              end--;
46
47      char hold;
48      while(begin < end)
49      {
50              hold = str[begin];
51              str[begin] = str[end];
52              str[end] = hold;
53
54              begin++;
55              end--;
56      }
57  };
58
59  void EncodeString(char str[],int len)
```

```
60  {
61      ReplaceChars(str, len);
62      ReverseString(str, len);
63  };
64
65  int main()
66  {
67      char hello[] = "hasd11H";
68
69      EncodeString(hello, strlen(hello));
70      cout << hello << endl;
71
72      return 0;
73  }
```

代码第 61～62 行，EncodeString()函数调用了 ReplaceChars ()函数和 ReverseString()函数，前者替代字符，后者翻转整个字符串。

ReplaceChars()函数调用GetFourthChar()函数来查找后面第四个字符并替换。GetFourthChar()函数的实现非常简单，它使用查找两个全局数组，这两个数组分别包含了所有的大小写字母。

ReverseString()函数里使用了 begin 和 end 两个分别指向字符串头尾的指针。然后头尾的内容不断交换，begin指针往尾移动，end指针往头移动，如此循环，直到两个指针碰头。

程序执行结果如下。

L11hwel

面试题20 反转字符串，但其指定的子串不反转

考点：字符串相关的综合编程能力

出现频率：★★★

给定一个字符串、一个这个字符串的子串，将第一个字符串反转，但保留子串的顺序不变。例如

输入： 第一个字符串: "Welcome you, my friend"

子串： "you"

输出： "dneirf ym ,you emocleW"

【解析】

对于本题，采取的一般步骤为：

（1）扫描一遍第一个字符串，然后用stack把它反转，同时记录下子串出现的位置。

（2）扫描一遍把记录下来的子串再用stack反转。

（3）将堆栈里的字符弹出，这样子串又恢复了原来的顺序。

这里使用一遍扫描数组的方法。扫描中如果发现子串，就将子串倒过来压入堆栈。最后将堆栈里的字符弹出，这样子串又恢复了原来的顺序。C++标准库中的stack表示一个后进先出的栈结构，使用stack可以轻松地完成这个转换。源程序如下。

```
1    #include <iostream>
2    #include <cassert>
3    #include <stack>
4    using namespace std;
5
```

```
6   const char* reverse(const char* s1, const char* token)
7   {
8          stack<char> stack1;
9          const char* ptoken = token, *head = s1, *rear = s1;
10       assert(s1 && token);
11       while (*head != '\0')
12       {
13              while(*head != '\0' && *ptoken == *head)
14              {
15                  ptoken++;
16                  head++;
17              }
18              if(*ptoken == '\0')
19              {
20                  const char* p;
21                  for(p=head-1; p>=rear; p--)
22                  {
23                         stack1.push(*p);
24                  }
25                  ptoken = token;
26                  rear = head;
27              }
28              else
29              {
30                  stack1.push(*rear++);
31                  head = rear;
32                  ptoken = token;
```

```
33              }
34          }
35          char *pReturn = new char[strlen(s1)+1];
36          int i=0;
37          while(!stack1.empty())
38          {
39              pReturn[i++] = stack1.top();
40              stack1.pop();
41          }
42          pReturn[i]='\0';
43
44          return pReturn;
45      }
46
47      int main(int argc, char* argv[])
48      {
49          char welcome[] = "Welcome you, my friend";
50          char token[] = "you";
52
51          const char *pRev = reverse(welcome, token);
53          cout << "before reverse:" << endl;
54          cout << welcome << endl;
55          cout << "after reverse:" << endl;
56          cout << pRev << endl;
57
58          return 0;
59      }
```

reverse()函数中，代码第13～17行搜索字符串的字串位置。其中指针ptoken记录当前扫描的子串位置，如果其内容为结束符'\0'（代码第 18～27 行），那么已经搜索到了这个子串，于是将它倒过来压入堆栈，否则直接压入堆栈（代码第 28～33 行）。最后申请一块堆内存，使用退栈把栈中内容存入堆内存中并返回堆内存。程序执行结果：

```
before reverse:
welcome you, my friend
after reverse:
dneirf ym ,you emocleW
```

面试题21 编写字符串反转函数strrev

考点：字符串相关的综合编程能力

出现频率：★★★★

编写字符串反转函数：strrev。要求时间和空间效率都尽量高。测试用例:输入"abcd"，输出应为"dcba"。

【解析】

看到这个题目，想到最简单、最直觉的解法就是：遍历字符串，将第一个字符和最后一个交换，第二个和倒数第二个交换，依次循环。于是有了解法1：

```
1   char* strrev1(const char* str)
2   {
3       int len = strlen(str);
4       char* tmp = new char[len + 1];
5
6       strcpy(tmp,str);
7       for (int i = 0; i < len/2; ++i)
8       {
9               char c = tmp[i];
10          tmp[i] = tmp[len - i - 1];
11          tmp[len - i - 1] = c;
12       }
13
14      return tmp;
15  }
```

解法1是通过数组下标的方式访问字符串字符的，也可以用指针直接操作。下面是解法2：

```
1   char* strrev2(const char* str)
2   {
3           char* tmp = new char[strlen(str) + 1];
4           strcpy(tmp,str);
5           char* ret = tmp;
6           char* p = tmp + strlen(str) - 1;
7
8           while (p > tmp)
9           {
10              char t = *tmp;
11                  *tmp = *p;
12                  *p = t;
13
14              --p;
15              ++tmp;
16        }
17
18        return ret;
19   }
```

解法 1 和解法 2 都没有考虑时间和空间的优化，一个典型的优化策略就是两个字符交换的算法优化，我们可以借助异或运算符（^）完成两个字符的交换，对应这里的解法3和解法4。解法3：

```
1   char* strrev3(const char* str)
2   {
3           char* tmp = new char[strlen(str) + 1];
```

```
4           strcpy(tmp,str);
5           char* ret = tmp;
6           char* p = tmp + strlen(str) - 1;
7
8           while (p > tmp)
9           {
10                  *p ^= *tmp;
11                  *tmp ^= *p;
12                  *p ^= *tmp;
13
14              --p;
15              ++tmp;
16        }
17
18        return ret;
19    }
```

解法4：

```
1    char* strrev4(const char* str)
2    {
3           char* tmp = new char[strlen(str) + 1];
4           strcpy(tmp,str);
5           char* ret = tmp;
6
7           char* p = tmp + strlen(str) - 1;
8
9           while (p > tmp)
10        {
```

```
11              *p = *p + *tmp;
12              *tmp = *p - *tmp;
13              *p = *p - *tmp;
14
15          --p;
16          ++tmp;
17      }
18
19      return ret;
20  }
```

我们还可以使用递归来解决这个问题。每次交换首尾两个字符，中间部分则又变为和原来字符串同样的问题。解法5：

```
1   char* reverse5(char* str,int len)
2   {
3       if (len <= 1)
4           return str;
5
6       char t = *str;
7       *str = *(str + len -1);
8       *(str + len -1) = t;
9
10      return (reverse5(str + 1,len - 2) - 1);
11  }
```

注意，这里5个解法中只有解法5修改了输入字符串，其他4种都是在函数体内申请堆内存。下面给出测试用的main()函数：

```
1   int main(int argc,char* argv[])
2   {
```

```
3       char* str = "123456";
4       cout << str << endl;
5
6       char* str2 = reverse1(str);
7       cout << str2 << endl;
8
9       char* str3 = reverse2(str2);
10      cout << str3 << endl;
11
12      char* str4 = reverse3(str3);
13      cout << str4 << endl;
14
15      char* str5 = reverse4(str4);
16      cout << str5 << endl;
17
18      char* str6 = reverse5(str5,strlen(str5));
19      cout << str6 << endl;
20
21      return 0;
22  }
```

程序输出结果：

```
1   123456
2   654321
3   123456
4   654321
5   123456
6   654321
```

面试题22 编程实现任意长度的两个正整数相加

考点：字符串相关的综合编程能力

出现频率：★★★★

【解析】

我们知道在C/C++中有int、float、double等类型来表示数字，但是它们的长度都是有限的。而本题要求可以是任意长度，这里可以用字符串表示数字，结果也用字符串表示。因此，我们所要做的就是做一个类似整数加法的字符串转换，主要是字符做加法运算并且要考虑进位。示例程序如下所示。

```
1    #include <stdio.h>
2    #include <string.h>
3    #include <stdlib.h>
4    #include <math.h>
5
6    char* addBigInt(char* num1, char* num2)
7    {
8          int c = 0;              //进位，开始最低进位为0
9          int i = strlen(num1)-1;    //指向第一个加数的最低位
10       int j = strlen(num2)-1;    //指向第二个加数的最低位
11       int maxLength = strlen(num1) >= strlen(num2) ?
12             strlen(num1)+1 : strlen(num2)+1;    //得到2个数中较大数
的位数
13       char* rst = (char*)malloc(maxLength+1);    //保存结果
```

```
14      int k;
15      if (rst == NULL)
16      {
17          printf("malloc error!\n");
18          exit(1);
19      }
20
21      rst[maxLength] = '\0';    //字符串最后一位为'\0'
22      k = strlen(rst) - 1 ;    //指向结果数组的最低位
23      while ( (i >= 0) && (j >= 0) )
24      {
25          rst[k] = ( (num1[i] - '0') + (num2[j] - '0') + c )%10 +'0'; //计算
本位的值
26          c = ( (num1[i] - '0') + (num2[j] - '0') + c )/10;            //向高位
进位值
27          --i;
28          --j;
29          --k;
30      }
31      while ( i >= 0 )
32      {
33          rst[k] = ( (num1[i] - '0') + c )%10 + '0';
34          c = ( (num1[i] - '0') + c)/10;
35          --i;
36          --k;
37      }
38      while ( j >= 0 )
```

```
39      {
40          rst[k] = ( (num2[j] - '0') + c )%10 + '0';
41          c = ( (num2[j] - '0') + c )/10;
42          --j;
43          --k;
44      }
45      rst[0] = c + '0';
46
47      if ( rst[0] != '0' )            //如果结果最高位不等于0，则输出
结果
48      {
49          return rst;
50      }
51      else
52      {
53          return rst+1;
54      }
55  }
56
57  int main()
58  {
59      char num1[] = "123456789323";
60      char num2[] = "45671254563123";
61      char *result = NULL;
62
63      result = addBigInt(num1, num2);
64      printf("%s + %s = %s\n", num1, num2, result);
```

```
65
66        return 0;
67    }
```

程序执行结果：

```
1    123456789323 + 45671254563123 = 45794711352446
```

面试题23 编程实现字符串的循环右移

考点：字符串相关的综合编程能力

出现频率：★★★★

【解析】

根据题意，我们编写的函数能把一个 char 组成的字符串循环右移 n 个。例如原来是"abcdefghi"，如果n=2，移位后应该是"hiabcdefgh"。

程序代码如下。

```
1    #include <stdio.h>
2    #include <stdlib.h>
3
4    void loopMove(char *str, int n)
5    {
6            int i = 0;
7            char *temp = NULL;
8            int strLen = 0;
9            char *head = str;              //指向字符串头
10
11         while(*str++);
12         strLen = str - head - 1;        //计算字符串长度
13         n = n % strLen;               //计算字符串尾部移到头部的字符个
数
14         temp = (char *)malloc(n);       //分配内存
15         for (i = 0; i < n; i++)
16         {
```

```
17              temp[i] = head[strLen - n + i]; //临时存放从尾部移到头部
的字符
18      }
19      for (i = strLen - 1; i >= n; i--)
20      {
21              head[i] = head[i - n];       //从头部字符移到尾部
22      }
23      for (i = 0; i < n; i++)
24      {
25              head[i] = temp[i];           //从临时内存区复制尾部字符
26      }
27
28      free(temp);
29  }
30
31  int main(void)
32  {
33      char string[] = "123456789";
34      int steps = 0;
35
36      printf("string: %s\n", string);
37      printf("input step: ");
38      scanf("%d", &steps);
39      loopMove(string, steps);    //向右循环移位
40      printf("after loopMove %d: %s\n", steps, string);
41
42      return 0;
```

43　}

程序执行结果：

```
1    string: 123456789
2    input step; 6（输入）
3    after loopMove 6: 456789123
```

程序中首先计算字符串尾部移到头部的字符个数，然后分配一个相同大小的堆内存来临时保存这些字符，最后做两次循环分别把头部字符移位到尾部，以及把堆内存中的内容复制到字符串头部。

面试题24 删除指定长度的字符

编程实现从字符串的指定位置开始，删除指定长度的字符。

考点：字符串相关的综合编程能力

出现频率：★★★★

【解析】

根据题意，假设一个字符串 "abcdefg"，如要从第二个开始（索引为1），删除两个字符，则删除后的字符串是"adefg"。

程序代码如下。

```
1    #include <stdio.h>
2    #include <string.h>
3
4    char *deleteChars(char *str,int pos,int len)
5    {
6        char *p = str + pos - 1;       //指向pos位置字符
7        int tt = strlen(str);          //计算字符串长度
8
9        if( (pos < 1) || (p-str) > tt) //检查pos是否不大于1
10       {                              //或者pos超出字符串长度
11           return str;
12       }
13
14       if( (p+len-str) > tt)          //len大于pos后剩余的字符个数
15       {                              //只需对pos位置赋'\0'
16           *p = '\0';
```

```
17              return str;
18          }
19
20          //删除len个字符
21          while(*p && *(p+len))           //len小于或等于pos后剩余的字
符个数
22          {                               //删除中间len个字符
23                  *p = *(p+len);
24              p++;
25          }
26            *p = '\0';
27
28          return str;
29      }
30
31      int main()
32      {
33          char string[] = "abcdefg";
34          int pos = 0;
35          int len = 0;
36          int steps = 0;
37          printf("string: %s\n", string);
38          printf("input pos: ");
39          scanf("%d", &pos);
40          printf("input len: ");
41          scanf("%d", &len);
42          deleteChars(string, pos, len);    //删除string的pos位置后len个字
```

符

43　　printf("after delete %d chars behind pos %d: %s\n", len, pos, string);

44

45　　return 0;

46　}

程序执行结果：

1　string: abcdefg

2　input pos: 2（输入）

3　input len: 2（输入）

4　after delete 2 chars behind pos 2: adefg

deleteChars函数首先检查传入pos以及len的合法性，然后找到字符串的pos位置，最后删除len个字符。

面试题25 字符串的排序及交换

考点：字符串相关的综合编程能力

出现频率：★★★

编写一个函数将一条字符串分成两部分，将前半部分按 ASCII 码升序排序，后半部分不变，（如果字符串是奇数，则中间的字符不变）再将前后两部分交换，最后将该字符串输出。测试字符串"ADZDDJKJFIEJHGI"。

【解析】

程序代码如下。

```
1    #include <stdio.h>
2    #include <stdlib.h>
3
4    /* 冒泡排序算法 */
5    void mysort(char *str, int num)
6    {
7        int i, j;
8        int temp = 0;
9
10       for (i = 0; i < num; i++)
11       {
12           for (j = i+1; j < num; j++)
13           {
14               if (str[i] < str[j])    //如果下一个值比当前值大，
15               {                       //则交换两个元素值
```

```
16                          temp = str[i];
17                          str[i] = str[j];
18                          str[j] = temp;
19                      }
20                  }
21          }
22      }
23
24      char *foo(char *str)
25      {
26          int len = 0;
27          char *start = NULL;
28
29          if (str == NULL)            //检查参数str的有效性
30          {
31                  return NULL;
32          }
33
34          start = str             //保存头部位置
35          while(*str++);
36          len = str - start - 1;    //计算字符串长度
37          len = len / 2;            //计算需要排序的字符个数
38          str = start;
39
40          mysort(str, len);           //从大到小排序
41
42          return str;
```

```
43  }
44
45  int main()
46  {
47      char string[] = "ADZDDJKJFIEJHGI";
48
49      printf("before transformation: %s\n", string);
50      foo(string);
51      printf("after transformation: %s\n", string);
52
53      return 0;
54  }
```

程序执行结果：

```
1  before transformation: ADZDDJKJFIEJHGI
2  after transformation: ZKJDDDAJFIEJHGI
```

foo()函数首先获得字符串的长度，然后计算需要排序的字符个数，最后调用mysort函数（使用冒泡排序方法）对字符进行排序。

面试题26 编程实现删除字符串中所有指定的字符

考点：字符串相关的综合编程能力

出现频率：★★★★

【解析】

根据题意，假设字符串为"cabcdefcgchci"，把该字符串中所有的字符'c'删除后，结果应该是"abdefghi"。

程序代码如下。

```
1    #include <stdio.h>
2
3    char *deleteChar(char *str,char c)
4    {
5           char *head = NULL;
6           char *p = NULL;
7
8           if (str == NULL)          //检查str的有效性
9           {
10               return NULL;
11        }
12
13        head = p = str;          //指向字符串头,准备遍历
14
15        while(*p++)
16        {
```

```
17              if(*p != c)       //如果不等于c的值，则在str中记录
18              {
19                   *str++ = *p;
20              }
21         }
22           *str = '\0';
23
24         return head;
25    }
26
27    int main(void)
28    {
29         char string[] = "cabcdefcgchci";
30         char c = 0;
31
32         printf("input char: ");
33         scanf("%c", &c);
34         printf("before delete: %s\n", string);
35         deleteChar(string, c);    //删除所有的c
36         printf("after delete: %s\n", string);
37
38         return 0;
39    }
```

程序执行结果：

```
1    input char: c（输入）
2    before delete: cabcdefcgchci
3    after delete: abdefghi
```

deleteChar()函数首先判断传入字符指针的有效性，然后使用两个指针进行操作。其中一个指针用来做记录，后一个指针进行遍历字符串。

面试题27 分析代码——使用strcat连接字符串

考点：字符串相关的综合编程能力

出现频率：★★★

下面的程序代码有什么问题吗？输出是什么？

```
1    #include<iostream>
2    using namespace std;
3
4    int main()
5    {
6        char *str1 = "hello";
7        char *str2 = " china";
8        char *str3 = NULL;
9
10       str3 = new char[strlen(str1)+strlen(str2)+1];
11       str3[0] = '\n';
12       strcat(str3,str1);
13       strcat(str3,str2);
14       cout << str3 << endl;
15
16       return 0;
17   }
```

【解析】

代码第12行和第13行调用strcat函数。strcat函数是库函数，其原型

如下。

```
1    extern char *strcat(char *dest, const char *src);
```

它把src字符串加到dest字符串之后，dest字符串结束符的位置就是新连接的src的位置。然而代码第10行中用new申请的堆内存是没有被初始化的，内存中的值都是随机数。代码第 12行调用 strcat不能把 str1的内容复制到堆内存块中，并且会导致数组越界。同样的问题发生在代码第13行调用中。应在代码第11行把str3[0]赋为结束符'\0'。正确的代码应为：

```
1    #include<iostream>
2    using namespace std;
3
4    int main()
5    {
6        char *str1 = "hello";
7        char *str2 = " china";
8        char *str3 = NULL;
9
10       str3 = new char[strlen(str1)+strlen(str2)+1];//分配堆内存
11       str3[0] = '\0';              //str3[0]赋为结束符'\0'，以便strcat能正常调用
12       strcat(str3,str1);           //str3变为"hello"
13       strcat(str3,str2);           //str3变为"hello china"
14       cout << str3 << endl;
15
16       return 0;
17   }
```

程序执行结果：

1　hello china

【答案】

原题中str3指向的堆内存没有初始化，不含有字符串结束符。输出是随机值。

面试题28 编程实现库函数strcat

考点：库函数strcat的实现细节

出现频率：★★★★★

【解析】

库函数strcat是把一个字符串内容连接到目标字符串的后面，所以应该从目标字符串的末尾，也就是结束符'\0'的位置插入另一个字符串的内容。

```
1    #include <stdio.h>
2    #include <stdlib.h>
3
4    char *mystrcat(char *dest, const char *src)
5    {
6            char *ret;
7
8            ret = dest;              //保存目的字符串首地址以便返回
9            while (*dest++);
10       dest--;                  //此时dest指向字符串结束符
11       while (*dest++ = *src++);    //循环复制
12
13       return ret;
14   }
15
16   int main(void)
17   {
```

```
18        char *dest = NULL;
19        char *str1 = "Hello ";
20        char *str2 = "World!";
21
23          *dest = '\0';                    //为把目标字符串置为空，将结束符放在其开头
22        dest = (char *)malloc(256);
24        mystrcat(mystrcat(dest, str1), str2);    //链式表达式连接str1和str2
25        printf("dest: %s\n", dest);
26        free(dest);
27        dest = NULL;
28
29        return 0;
30    }
```

程序执行结果：

```
1    Hello World!
```

面试题29 编程计算含有汉字的字符串长度

考点：字符串相关的综合编程能力

出现频率：★★

编写gbk_strlen函数，计算含有汉字的字符串的长度，汉字作为一个字符处理；已知：汉字编码为双字节，其中首字节<0，尾字节在0～63以外（如果一个字节是−128～127）。

【解析】

程序代码如下。

```
1    #include <iostream>
2    using namespace std;
3
4    int gbk_strlen(const char *str)
5    {
6            const char *p = str;        //p用于后面遍历
7
8            while(*p)                //若是结束符0，则结束循环
9            {
10                if (*p < 0 && (*(p+1)<0 || *(p+1)>63))    //中文汉字情况
11                {
12                    str++;        //str移动一位，p移动两位，因此长度加1
13                    p += 2;
14                }
15                else
16                {
```

```
17              p++;          //str不动，p移动一位，长度加1
18          }
19      }
20
21      return p-str;          //返回地址之差
22  }
23
24  int main()
25  {
26      char str[] = "abc你好123中国456";       //含有中文汉字的字符
串
27
28      int len = gbk_strlen(str);              //获得字符串长度
29      cout << str << endl;
30      cout << "len = " << len << endl;
31
32      return 0;
33  }
```

gbk_strlen()函数中，使用了两个指针指向的地址之差来获得字符串长度。当遇到中文汉字时，由于中文汉字占两个字节，因此 p 移动两个指向中文汉字的后一个字符；而同时为了使汉字的长度算1个，则需要将src移动一位。程序执行结果如下。

```
1   abc你好123中国456
2   len = 13
```

面试题30 找出01字符串中0和1连续出现的最大次数

考点：字符串相关的综合编程能力

出现频率：★★

【解析】

程序代码如下。

```
1    #include <iostream>
2    using namespace std;
3
4    void Calculate(const char *str, int *max0, int *max1)
5    {
6        int temp0 = 0;    //保存连续是'0'的最大长度
7        int temp1 = 0;    //保存连续是'1'的最大长度
8
9        while(*str)       //遍历01字符串
10       {
11               if (*str == '0') //当前字符是'0'
12               {
13                   (*max0)++;               //'0'的计算长度加1
14                   if (*(++str) == '1')    //如果下一个字符是'1'
15                   {
16                           if (temp0 < *max0)    //判断当前最大长度是否需
要保存
17                           {
```

```
18                  temp0 = *max0;
19                }
20                  *max0 = 0;
21            }
22        }
23        else if (*str == '1') //当前字符是'1'
24        {
25            (*max1)++;          //'1'的计算长度加1
26            if (*(++str) == '0')   //如果下一个字符是'0'
27            {
28                if (temp1 < *max1) //判断当前最大长度是否需要
保存
29                {
30                    temp1 = *max1;
31                }
32                    *max1 = 0;
33            }
34        }
35    }
36
37    *max0 = temp0;            //'0'的最大长度返回max0
38    *max1 = temp1;            //'1'的最大长度返回max1
39  }
40
41  int main(int argc, char *argv[])
42  {
43    char string[] =
```

```
"00001110110000001100110101101001010101011111010";
    44
    45        int max0 = 0;
    46        int max1 = 0;
    47
    48        Calculate(string, &max0, &max1);    //计算max0和max1
    49        cout << "max0 = " << max0 << endl;
    50        cout << "max1 = " << max1 << endl;
    51
    52        return 0;
    53    }
```

程序输出结果：

```
1    max0 = 6
2    max1 = 5
```

面试题31 编程实现字符串的替换

考点：字符串相关的综合编程能力

出现频率：★★★

用 C++写一个小程序，先请用户输入 3 个字符串，然后把在第一个字符串中出现的所有第2个字符串替换成第3个字符串，最后输出新的字符串。

【解析】

程序代码如下。

```
1    #include <iostream>
2    using namespace std;
3
4    char *replace(const char *str, const char *sub1, const char *sub2,
char *output)
5    {
6        char *pOutput = NULL;
7        const char *pStr = NULL;
8        int lenSub1 = strlen(sub1);    //子串sub1的长度
9        int lenSub2 = strlen(sub2);    //子串sub2的长度
10
11        pOutput = output;
12        pStr = str;                  //用于寻找子串
13        while(*pStr != 0)
14        {
15            pStr = strstr(pStr, sub1);//在str中寻找sub1子串
```

```
16          if (NULL != pStr)       //找到sub1子串
17          {
18              memcpy(pOutput, str, pStr-str);   //复制str的前一部分
output
19              pOutput += pStr-str;
20              memcpy(pOutput, sub2, lenSub2);   //复制sub2子串到
output
21              pOutput += lenSub2;
22              pStr += lenSub1;        //为了下一次复制做准备
23              str = pStr;
24          }
25          else                    //找不到sub1子串
26          {
27              break;
28          }
29      }
30      *pOutput = '\0';
31      if (*str != '\0')
32      {
33          strcpy(pOutput, str);   //复制str剩余部分到ouput
34      }
35
36      return output;
37  }
38
39  int main()
40  {
```

```
41      char str[50] = "";               //源字符串str
42      char sub1[10]    = "";           //被替换的字符串sub1
43      char sub2[10]    = "";           //用来替换sub2
44      char output[100] = "";           //输出字符串
45
46      cout << "str: ";
47      cin >> str;
48      cout << "sub1: ";
49      cin >> sub1;
50      cout << "sub2: ";
51      cin >> sub2;
52
53      cout << replace(str, sub1, sub2, output) << endl;
54
55      return 0;
56  }
```

程序执行结果：

```
1   str: abcdefcdg
2   sub1: cd
3   sub2: 123
4   ab123ef123g
```

replace()函数把str中的所有sub1子串替换为sub2子串，结果存入output指向的内存块中。它循环使用了库函数strstr寻找字符串中的子串，如果找到sub1子串，就往output复制str中不属于sub1子串的部分以及sub2字符串。如果找不到sub1子串，则退出循环，并且把str剩余的部分复制到output。

第5章 位运算与嵌入式编程

C 语言是嵌入式开发所必需的编程技术，因此在招聘嵌入式系统程序员时，它是一种非常有效的方式。笔者参加过许多相关的测试，并发现这些测试能为应试者与面试者提供许多有用的信息。虽然在应试的过程中会有一些压力，但是这些测试本身也十分有趣。

对于应试者来说，可以通过测试题了解出题者的一些情况。例如出题者是擅长微机还是嵌入式系统。对于面试者来说，一个测试题能从多个方面反映应试者的素质，最基本的是 C 语言编程的水平。如果应试者不会这道测试题，那么他采取的回答方式可以反映出他的一些基本素质。他是找各种借口呢，还是表现出学习的好奇心？这些信息往往与应试者的测试成绩一样有用。本章列出了一些针对嵌入式系统的面试题，希望能给正在找工作的人一些帮助。

面试题1 位制转换

写出下面代码的输出结果。

考点：使用printf输出不同类型的变量

出现频率：★★★★

```
1    #include <stdio.h>
2
3    int main(int argc, char* argv[])
4    {
5        int i = 5.0l;
6        float f = 5;
7
8        printf("%f\n", 5);
9        printf("%lf\n", 5.0l);
10       printf("%f\n", f);
11
12       printf("%d\n", 5.0l);
13       printf("%d\n", i);
14
15       return 0;
16   }
```

【解析】

32位平台中int和float都占4个字节，double占8个字节。以下的讨论都是基于32位平台的。

printf根据说明符“%f”，认为参数应该是个double类型的参数（在

printf函数中，float会自动转换成double），因此从栈中读了8个字节。类似地，当printf后的说明符为"%d"时，会认为参数应该是个int类型的参数，因此从栈中读了4个字节。

代码第8行，首先参数5为int型，所以在栈中分配了4个字节的内存用于存放参数5。然后printf从栈中读8个字节。很显然，内存访问越界，会有不可预料的情况发生。

代码第9行和第10行，参数5.0l和f分别是double类型和float类型（注意，这里f在赋值时已经做了一次int到float的转化），而float和double都是浮点类型，其相互转化是相对安全的，因此这两段代码输出都是5.000000。

代码段printf("%d\n", 5.0l);，参数5.0l 占8 个字节，而printf读4 个字节，同样会出现不可预料的情况。

代码段printf("%d\n", i);，参数i 是由5.0l 转换过来的，这里的转换不同于printf 中的读取，在数据大小允许的范围之内（没有溢出），转换是安全的。转换的结果是i等于5，占4个字节，因此这段代码输出为5。

从上述分析中可以看出，如果在printf或者scanf中指定了"%f"，那么后面对应的参数列表也应该是浮点数，或者是一个指向浮点变量的指针，否则就不应该加载支持浮点数的函数。

【答案】

```
1    0.000000
2    5.000000
3    5.000000
4    0
5    5
```

面试题2 看代码写出结果——位运算

考点：使用位操作符>>和<<

出现频率：★★★★

```
1    #include <stdio.h>
2
3    int main(int argc, char* argv[])
4    {
5        unsigned short int i = 0;
6        unsigned char ii = 255;
7        int j = 8, p, q;
8
9        p = j << 1;
10         q = j >> 1;
11         i = i – 1;
12         ii = ii + 1;
13
14         printf("i = %d\n", i);
15         printf("ii = %d\n", ii);
16         printf("p = %d\n", p);
17         printf("q = %d\n", q);
18
19         return 0;
20    }
```

【解析】

本题有两个考点：一是无符号结果的问题；二是用移位的方式代替乘除法。

变量i 是一个unsigned short int 类型，在32 位平台下大小是2 个字节，因此其无符号类型的大小范围为0～65 535。i 赋值之后为0，内存中的数据为0x0000。当执行了i = i – 1;之后，内存中的数据变为0Xffff，所以结果就是65 535。

变量ii 是一个unsigned short char类型，大小是1 个字节，因此其无符号类型的大小范围为0～255。ii 赋值之后为255，内存中的数据为0xff。当执行了ii = ii + 1;之后，内存中的数据变为0X00，所以结果就是0。

左移操作<<相当于乘法操作，<< n相当于乘以2n。右移操作>>相当于除法操作，>> n相当于除以2n。因此，对于变量p和q的输出分别是16和4。

通过以上分析可以看出，对于无符号结果的问题，我们需要十分清楚数据在内存中的大小以及其表达形式。另外，我们需要知道移位操作是最有效率的（可以代替乘除法），因此在嵌入式系统编程或者其他许多需要高效率的地方能够得到运用。

【答案】

1　i = 65535

2　ii = 0

3　p = 16

4　q = 4

面试题3 设置或清除特定的位

考点：使用位操作符&和|

出现频率：★★★★

嵌入式系统总是要求用户对变量或寄存器进行位操作。给定一个整型变量a，写两段代码，第一个设置a 的bit 3，第二个清除a 的bit 3。在以上两个操作中，要保持其他位不变。

【解析】

通常情况下，应试者对这个问题有3种基本的反应：

（1）不知道如何下手。该被面试者从没做过任何嵌入式系统的工作。

（2）用bit fields。bit fields是被扔到C语言死角的东西，它保证你的代码在不同编译器之间是不可移植的，同时也保证了你的代码是不可重用的。

（3）用#defines 和 bit masks 操作。这是一个有极高可移植性的方法，是应该被用到的方法。

最佳的解决方案为：

```
1   #define BIT3 (0x1 << 3)
2   static int a;
3
4   void set_bit3(void)
5   {
6           a |= BIT3;
7   }
8
```

```
9    void clear_bit3(void)
10   {
11        a &=～BIT3;
12   }
```

在这里，BIT3用来计算需要操作的位，|=和&=分别用来指定位置1和位置0。

面试题4 计算一个字节里有多少bit被置1

考点：各种位操作符的使用

出现频率：★★★

【解析】

源程序如下所示。

```
1    #include <stdio.h>
2
3    #define BIT7 (0x1 << 7)
4
5    int calculate(unsigned char c)
6    {
7            int count = 0;
8            int i = 0;
9            unsigned char comp = BIT7;
10
11        for (i = 0; i < sizeof(c) * 8; i++)
12        {
13                if ((c & comp) != 0)
14                {
15                        count++;
16                }
17                comp = comp >> 1;
18        }
19
```

```
20      return count;
21  }
22  int main(int argc, char* argv[])
23  {
24      unsigned char c = 0;
25      int count = 0;
26
27      printf("c = ");
28      scanf("%d", &c);
29
30      count = calculate(c);
31      printf("count = %d\n", count);
32      return 0;
33  }
```

程序说明：

一个字节（byte）有8位，因此首先在宏定义BIT7中将最高位置成1。然后在calculate函数中比较每个位是否被置成1，如果是，则为count++，循环结束后返回count的值。最后在 main 函数中进行测试：如果输入的值为 97（二进制为 1100001），则最后打印出 count= 3。

面试题5 位运算改错

考点：正确使用位运算符和逻辑运算符

出现频率：★★★

```
1   // The function need set corresponding bit into 0
2   #define BIT_MASK(bit_pos) (0x01<<(bit_pos))
3
4   int Bit_Reset(unsigned int* val, unsigned char pos)
5   {
6           if (pos >= sizeof(unsigned int) * 8)
7           {
8               return 0;
9           }
10      val = (val &&～BIT_MASK(pos));
11      return 1;
12  }
```

【解析】

Bit_Reset函数的作用是要把相应的位置0。首先是位掩码BIT_MASK(bit_pos)的定义，它的需要置0的位是1，其他位都是0。然后在Bit_Reset函数中判断pos，如果超出整型的字节范围，则表示返回0 失败。最后调用val = (val && ～BIT_MASK(pos))；，代码段将val的pos位置0。

这里存在位运算的问题。在最后的置0操作中用的是"&&"操作符，它表示的是“逻辑与”，其结果为只要val不为0，调用完第10行代码之后val都变成1，否则便为0。这与设计函数的初衷不符，应该将"&"替换

为"&&"。

面试题6 运用位运算交换a、b两数

考点：位运算的灵活使用

出现频率：★★★

【解析】

源程序如下所示。

```
1    #include <stdio.h>
2
3    int main()
4    {
5        int a = 3;
6        int b = 5;
7
8        a ^= b;                          //进行三次异或操作
9        b ^= a;
10       a ^= b;
11
12       printf("a = %d, b = %d\n", a, b);    //打印a、b值
13       return 0;
14   }
```

^是位异或的运算符，即比较相同两位的异同，如果相同，则赋值为0，否则为1。

此例中a、b的初始值分别为3和5，对应的二进制分别为00000011和00000101。经过下面的3个步骤交换了两个变量的值。

代码第 8 行，a 的二进制变为 00000110，b 仍为 00000101。即 b 不

变，取出所有不相等的位存入a。

代码第9 行，a 的二进制为00000110，b 变为00000011。即a不变，取出所有不相等的位存入b。此时b的值为a的初始值。

代码第10 行，a 的二进制变为00000101，b 为00000011。即b不变，取出所有不相等的位存入a。此时a的值为b的初始值。到此完成a、b两变量值的变换。

算法最大的优点是省略了中间变量，但只能用于相同类型数的交换。程序执行结果：

```
1    a = 5, b = 3
```

面试题7 列举并解释C++中的4种运算符转化以及它们的不同点

考点：位运算的灵活使用

出现频率：★★★★

【解析】

4种运算符如下。

（1）const_cast 操作符：用来帮助调用那些应该使用却没有使用const 关键字的函数。换句话说，就是供程序设计师在特殊情况下将限制为const成员函数的const定义解除，使其能更改特定属性。

（2）dynamic_cast 操作符：如果启动了支持运行时间类型信息（RTTI），dynamic_cast可以有助于判断在运行时所指向对象的确切类型。它与 typeid 运算符有关。可以将一个基类的指针指向许多不同的子类型（派生类），然后将被转型为基础类的对象还原成原来的类。不过，限于对象指针的类型转换，而非对象变量。

（3）reinterpret_cast操作符：将一个指针转换成其他类型的指针，新类型的指针与旧指针可以毫不相关。通常用于某些非标准的指针数据类型转换，例如将void *转换为char *。它也可以用在指针和整型数之间的类型转换上。注意：它存在潜在的危险，除非有使用它的充分理由，否则就不要使用它。例如，它能够将一个int *类型的指针转换为float *类型的指针，但是这样就会很容易造成整数数据不能被正确地读取。

（4）static_cast操作符：它能在相关的对象和指针类型之间进行类型转换。有关的类之间必须通过继承、构造函数或者转换函数发生联系。static_cast操作符还能在数字（原始的）类型之间进行类型转换。通常情况下，static_cast操作符大多用于将数域宽度较大的类型转换为较小

的类型。当转换的类型是原始数据类型时，这种操作可以有效地禁止编译器发出警告。

面试题8 用#define声明一个常数

用#define声明一个常数，用以表明1年中有多少秒？

考点：#define的使用规范

出现频率：★★★★

【解析】

源代码如下。

```
1    #define SECONDS_PER_YEAR (60 * 60 * 24 * 365)UL
```

面试官在这里想看到几件事情：

（1）#define语法的基本知识（例如，不能以分号结束，括号的使用，等等）；

（2）懂得预处理器将为你计算常数表达式的值，因此，直接写出你是如何计算 1 年中有多少秒而不是计算出实际的值，是更清晰而没有代价的；

（3）意识到这个表达式将使一个16位机的整型数溢出，因此要用到长整型符号L，告诉编译器这个常数是长整型数；

（4）如果你在你的表达式中用到UL（表示无符号长整型），那么你有了一个好的起点。记住，第一印象很重要。

面试题9 如何用C语言编写死循环

考点：死循环的编写方式

出现频率：★★★★

【解析】

这个问题有3个解决方案。

（1）首选的方案是：

```
while(1) { }
```

（2）一些程序员更喜欢如下方案：

```
for(;;) { }
```

这个实现方式会让面试官为难，因为这个语法没有确切地表达出到底是怎么回事。如果一个应试者把这个作为方案，面试官可能将用这个作为一个机会去探究应试者这样做的基本原理。如果应试者的基本答案是："我被教着这样做，但从没有想到过为什么。"这会给面试官留下一个坏印象。

（3）第3个方案是用goto：

```
Loop:
…
goto Loop;
```

应试者如给出上面的方案，这说明他是一个汇编语言程序员（这也许是好事）或者他是一个想进入新领域的BASIC/FORTRAN程序员。

面试题10 如何访问特定位置的内存

考点：合理编写代码，访问特定内存

出现频率：★★★★

嵌入式系统经常要求程序员去访问某特定的内存位置的特点。在某工程中，要求设置一绝对地址为0x67a9的整型变量的值为0xaa66。编译器是一个纯粹的ANSI编译器。写代码去完成这一任务。

【解析】

源代码如下。

```
1    int *ptr;
2    ptr = (int *)0x67a9;
3    *ptr = 0xaa55
```

当然，还可以用下面比较晦涩的方法。

```
1    *(int * const)(0x67a9) = 0xaa55;
```

即使你的品味更接近第二种方案，但建议你在面试时使用第一种方案。

面试题11 对中断服务代码的评论

考点：对嵌入式系统中断服务的理解

出现频率：★★★

中断是嵌入式系统中重要的组成部分，这导致很多编译开发商提供一种扩展，让标准C支持中断。而事实是，产生了一个新的关键字__interrupt。下面的代码就使用了__interrupt关键字去定义一个中断服务子程序（ISR），请评论一下这段代码。

```
1   __interrupt double compute_area (double radius)
2   {
3       double area = PI * radius * radius;
4       printf(" Area = %f", area);
5       return area;
6   }
```

【解析】

这个函数有以下方面的错误。

（1）ISR不能返回一个值。如果你不懂这个，那么你是不会被雇用的。

（2）ISR不能传递参数。如果你没有看到这一点，那么你被雇用的机会等同于第1点。

（3）在许多的处理器/编译器中，浮点一般都是不可重入的。有些处理器/编译器需要让额外的寄存器入栈，有些处理器/编译器就是不允许在ISR 中做浮点运算。此外，ISR应该是短且有效率的，在ISR中做浮点运算是不明智的。

（4）与第3点一脉相承，printf()经常有重入和性能上的问题。如果

你丢掉了第3点和第4点，面试官也不会太为难你的。不用说，如果你能得到后两点，那么你的被雇用前景越来越光明了。

面试题12 看代码写结果——整数的自动转换

考点：对C语言中整数自动转换原则的理解

出现频率：★★★

```
1    Void foo(void)
2    {
3        unsigned int a = 6;
4        int b = -20;
5        if (a+b > 6)
6                puts("> 6");
7        else
8                puts("<= 6");
9    }
```

【解析】

这个问题测试你是否懂得 C 语言中的整数自动转换原则，有些应试者极少懂得这些东西。不管如何，这无符号整型问题的答案输出是“>6”。原因是当表达式中存在有符号类型和无符号类型时，所有的操作数都自动转换为无符号类型。因此，−20变成了一个非常大的正整数，所以该表达式计算出的结果大于6。这一点对于频繁用到无符号数据类型的嵌入式系统来说是非常重要的。如果你答错了这个问题，你也就与这份工作无缘了。

面试题13 关键字static的作用是什么

考点：对C语言中关键字static作用的理解

出现频率：★★★★★

【解析】

这个简单的问题很少有人能完全回答。在C语言中，关键字static有以下3个明显的作用。

（1）在函数体内，一个被声明为静态的变量在这一函数被调用的过程中维持其值不变。

（2）在模块内（但在函数体外），一个被声明为静态的变量可以被模块内所有函数访问，但不能被模块外其他函数访问。它是一个本地的全局变量。

（3）在模块内，一个被声明为静态的函数只可被这一模块内的其他函数调用。那就是，这个函数被限制在声明它模块的本地范围内使用。

大多数应试者能正确回答第 1 部分，另一部分应试者能正确回答第 2 部分，同时很少的人能懂得第 3 部分。这是应试者的一个严重的不足，因为他显然不懂得本地化数据和代码范围的好处及重要性。

面试题14 关键字volatile有什么含义

考点：对C语言中关键字volatile作用的理解

出现频率：★★★

【解析】

一个定义为volatile的变量是说这变量可能会被意想不到地改变，这样，编译器就不会去假设这个变量的值了。精确地说，就是优化器在用到这个变量时必须每次都小心地重新读取这个变量的值，而不是使用保存在寄存器里的备份。下面是volatile变量的几个例子。

（1）并行设备的硬件寄存器（如状态寄存器）；

（2）一个中断服务子程序中会访问到的非自动变量（Non-automatic variables）；

（3）多线程应用中被几个任务共享的变量。

回答不出这个问题的人是不会被雇用的。这是区分 C 语言程序员和嵌入式系统程序员的最基本的问题。嵌入式系统程序员经常和硬件、中断、RTOS等打交道，所有这些都要求懂得volatile变量。不懂得volatile内容将会带来灾难。

面试题15 判断处理器是Big_endian还是Little_endian

编写函数，判断处理器是Big_endian还是Little_endian。

考点：对处理器字节序的理解以及union的作用

出现频率：★★★

【解析】

这里编写一个函数，若处理器是Big_endian的，则返回0；若是Little_endian的，则返回1。

源程序如下。

```
1   int checkCPU()
2   {
3           union w
4           {
5               int a;
6               char b;
7           } c;
8           c.a = 1;
9           return (c.b == 1);
10  }
```

嵌入式系统开发者应该对Little-endian和Big-endian模式非常了解。采用Little-endian模式的CPU对操作数的存放方式是从低字节到高字节，而Big-endian模式对操作数的存放方式是从高字节到低字节。例如，16bit的数0x1234在Little-endian模式CPU内存中的存放方式（假设从地址0x4000开始存放）为：

0x4000:　　0x34

0x4001:　　0x12

而在Big-endian模式CPU内存中的存放方式则为：

0x4000:　　0x12

0x4001:　　0x34

32bit 宽的数 0x12345678 在 Little-endian 模式 CPU 内存中的存放方式（假设从地址0x4000开始存放）为：

0x4000:　　0x78

0x4001:　　0x56

0x4000:　　0x34

0x4001:　　0x12

而在Big-endian模式CPU内存中的存放方式则为：

0x4000:　　0x12

0x4001:　　0x34

0x4000:　　0x56

0x4001:　　0x78

联合体union的存放顺序是所有成员都从低地址开始存放，利用该特性，轻松地获得了CPU对内存采用Little-endian还是Big-endian模式的读写。

面试题16 评价代码片断——处理器字长

考点：处理器字长的认识

出现频率：★★★

```
unsigned int zero = 0;
unsigned int compzero = 0xFFFF;
/*1's complement of zero */
```

【解析】

对于一个int型不是16位的处理器来说，上面的代码是不正确的。应编写如下：

```
unsigned int compzero = ～0;
```

这一问题能真正揭露出应试者是否懂得处理器字长的重要性。好的嵌入式程序员能够非常准确地明白硬件的细节和它的局限性，然而PC机程序员往往把硬件作为一个无法避免的烦恼。

第6章 C++面向对象

C语言是面向过程的，而C++作为C语言的超集支持，它是面向对象的编程。面向对象（Object Oriented）是当前计算机界关心的重点，它是当今软件开发方法的主流，因此也是各大公司的重要考点。在面试过程中，求职者应该对面向对象的基本概念、类、对象构造函数以及多态性等有清晰的认识，具备编程实现面向对象各个方面功能的能力。

面试题1 描述面向对象技术的基本概念

考点：面向对象的基本概念

出现频率：★★★★

【解析】

面向对象的基本概念：按照人们认识客观世界的系统思维方式，采用基于对象（实体）的概念建立模型，模拟客观世界分析、设计、实现软件的办法。通过面向对象的理念使计算机软件系统能与现实世界中的系统一一对应。它包括下面几方面的内容。

类（class）：具有相似的内部状态和运动规律的实体集合。类来自于人们认识自然、认识社会的过程。在这一过程中，人们主要使用两种方法：由特殊到一般的归纳法和由一般到特殊的演绎法。在归纳的过程中，从一个个具体的事物中把共同的特征抽取出来，形成一个一般的概念。在演绎的过程中又把同类的事物，根据不同的特征分成不同的小类。

对象（object）：指现实世界中各种各样的实体，也就是类（class）的实例。它既可以指具体的事物，也可以指抽象的事物。每个对象都有自己的内部状态和运动规律。在面向对象概念中，把对象的内部状态称为属性，运动规律称为方法或事件。

消息（message）：指对象间相互联系和相互作用的方式。一个消息主要由5部分组成：发送消息的对象、接收消息的对象、消息传递办法、消息内容（参数）、反馈。

类的特性：抽象、继承、封装、重载、多态。

【答案】

面向对象是指按人们认识客观世界的系统思维方式，采用基于对象

（实体）的概念建立模型，模拟客观世界分析、设计、实现软件的办法，包括类、对象、消息以及类的特性等方面的内容。

面试题2 判断题——类的基本概念

考点：面向对象的基本概念

出现频率：★★★★

Which is incorrect about the class?（对于类，下面哪一个是不正确的？）

A．A class is a blueprint to objects.

B．We use the keyword class to create a class construct.

C．Once a class is declared, the class name becomes a type name and can be used to declarevariables.

D．The class is same as the struct，and there are no difference between class and struct.

【解析】

A．一个类是对象的设计蓝图，正确。因为对象是类的实例，只有类设计好了，对象才可以被创建。

B．使用class关键字创建一个类的结构，正确。class英文就是“类型”的意思。不仅在C++中使用class创建类，而且在Java和C#里都使用class关键字创建类。

C．一个类一旦被声明了，这个类名就成为一个类型名并可以使用它来声明变量，正确。

D．class与struct类似，它们之间没有任何不同，错误。

【答案】

D

面试题3 选择题——C++与C语言相比的改进

考点：C++对于C语言的改进点

出现频率：★★★★

C++是从早期的C语言逐渐发展演变来的。与C语言相比，它在求解问题的方法上进行的最大改进是什么？

A．面向过程

B．面向对象

C．安全性

D．复用性

【解析】

C++是从C语言发展演变来的。C语言是过程式编程语言，它以过程为中心、以算法为驱动。而 C++能够使用面向对象的编程方式，即使用以对象为中心、以消息为驱动的编程方式。这是C++在C语言上的最大改进。

【答案】

B

面试题4 class和struct有什么区别

考点：class和struct的区别

出现频率：★★★★

【解析】

这里有两种情况下的区别：

（1）C语言的struct与C++的class的区别。

（2）C++中的struct和class的区别。

在第一种情况下，struct与class有着非常明显的区别。C是一种过程化的语言，struct只是作为一种复杂数据类型定义，struct中只能定义成员变量，不能定义成员函数。例如下面的C代码片断。

```
1   struct Point
2   {
3           int x; //合法
4           int y; //合法
5           void print()
6           {
7               printf("Point print\n"); //编译错误
8           }
9   }
```

这里第7行会出现编译错误，提示如下的错误消息："函数不能作为Point结构体的成员"。因此大家看到在第一种情况下，struct 只是一种数据类型，不能使用面向对象编程。现在来看第二种情况。首先请看下面的代码。

```
1   #include <iostream>
```

```
2   using namespace std;
3
4   class CPoint
5   {
6         int x;                  //默认为private
7         int y;                  //默认为private
8         void print()                //默认为private
9         {
10          cout << "CPoint: (" << x << ", " << y << ")" << endl;
11       }
12    public:
13       CPoint(int x, int y)       //构造函数，指定为public
14       {
15          this->x = x;
16          this->y = y;
17       }
18       void print1() //public
19       {
20          cout << "CPoint: (" << x << ", " << y << ")" << endl;
21       }
22    };
23
24    struct SPoint
25    {
26       int x;              //默认为public
27       int y;              //默认为public
28       void print()           //默认为public
```

```
29      {
30          cout << "SPoint: (" << x << ", " << y << ")" << endl;
31      }
32      SPoint(int x, int y)    //构造函数,默认为public
33      {
34          this->x = x;
35          this->y = y;
36      }
37  private:
38      void print1()    //private类型的成员函数
39      {
40          cout << "SPoint: (" << x << ", " << y << ")" << endl;
41      }
42  };
43
44  int main(void)
45  {
46      CPoint cpt(1, 2);        //调用CPoint带参数的构造函数
47      SPoint spt(3, 4);        //调用SPoint带参数的构造函数
48
49      cout << cpt.x << " " << cpt.y << endl;    //编译错误
50      cpt.print();                //编译错误
51      cpt.print1();            //合法
52
53      spt.print();                    //合法
54      spt.print1();                    //编译错误
55      cout << spt.x << " " << spt.y << endl;    //合法
```

```
56
57      return 0;
58  }
```

在上面的程序里，struct还有构造函数和成员函数，其实它还拥有class的其他特性，例如继承、虚函数等。因此，C++中的struct扩充了C的struct功能。那么它们有什么不同呢？

main函数内的编译错误全部是因为访问private成员而产生的。因此我们可以看到class中默认的成员访问权限是private的，而struct中则是public的。在类的继承方式上，struct和class又有什么区别？请看下面的程序。

```
1   #include <iostream>
2   using namespace std;
3
4   class CBase
5   {
6   public:
7       void print()                //public成员函数
8       {
9           cout << "CBase: print()..." << endl;
10      }
11  };
12
13  class CDerived1 : CBase             //默认private继承
14  {
15  };
16
17  class CDerived2 : public Cbase      //指定public继承
```

```
18  {
19  };
20
21  struct SDerived1 : Cbase            //默认public继承
22  {
23  };
24
25  struct SDerived2 : private Cbase    //指定public继承
26  {
27  };
28
29  int main()
30  {
31      CDerived1 cd1;
32      CDerived2 cd2;
33      SDerived1 sd1;
34      SDerived2 sd2;
35
36      cd1.print();            //编译错误
37      cd2.print();
38      sd1.print();
39      sd2.print();            //编译错误
40
41      return 0;
42  }
```

可以看到，以private方式继承父类的子类对象不能访问父类的public成员。class继承默认是private继承，而struct继承默认是public继

承。

另外，在C++模板中，类型参数前面可以使用class或typename。如果使用struct，则含义不同，struct 后面跟的是"non-type template parameter"，而class 或typename 后面跟的是类型参数。

事实上，C++中保留struct的关键字是为了使C++编译器能够兼容C语言开发的程序。

【答案】

分以下两种情况。

C 语言的struct 与C++的class 的区别：struct 只是作为一种复杂数据类型定义，不能用于面向对象编程。

C++中的 struct 和 class 的区别：对于成员访问权限以及继承方式，class 中默认的是private的，而struct中则是public的。class还可以用于表示模板类型，struct则不行。

面试题5 改错——C++类对象的声明

考点：C++类对象的声明方法

出现频率：★★★

```
1    struct Test
2    {
3            Test( int ) {}
4            Test() {}
5            void fun() {}
6    };
7
8    void main( void )
9    {
10       Test a(1);
11       a.fun();
12       Test b();
13       b.fun();
14    }
```

【答案】

题中的 Test 有两个构造函数，其中一个是带参数的，而另一个是不带参数的。在调用不带参数的构造函数时不需要加小括号，因此代码第12行是错误的。应该改成：

```
1    Test b; //去掉小括号
```

面试题6 看代码写结果——C++类成员的访问

考点：类对象的私有成员函数不能用对象访问

出现频率：★★★

```
1    #define public private       // (1)
2
3    class Animal
4    {
5    public:                  //    (2)
6            void MakeNoise();
7    };
8
9    int main(void)
10   {
11       Animal animal;
12       animal.MakeNoise();       // (3)
13       return 0;
14   }
```

A．（1） B．（2） C．（3） D．（1）（2）（3）

【解析】

（1）正确。把public宏定义为private。

（2）正确。定义 public 成员。注意，由于 public 已经被定义为private，因此这里的MakeNoise()成员函数实际上是private的。

（3）错误。调用Animal对象的私有成员函数。

【答案】

C

面试题7 找错——类成员的初始化

考点：初始化列表的构造顺序

出现频率：★★★

```
1    #include <iostream>
2    using namespace std;
3
4    class Obj {
5    public:
6            Obj(int k) : j(k), i(j)
7            {
8            }
9            void print(void)
10         {
11             cout << i << endl << j << endl;
12         }
13    private:
14         int i;
15         int j;
16    };
17
18    int main(int argc, char *argv[])
19    {
20         Obj obj(2);
21         obj.print();
```

```
22
23        return 0;
24    }
```

【解析】

本题考查的是初始化列表方面的知识。这里很容易让人以为先用2对j进行初始化，然后用j对i进行初始化，那么i和j都是2。

实际上，初始化的顺序正好与想象中的相反。

初始化列表的初始化顺序与变量声明的顺序一致，而不是按照出现在初始化列表中的顺序。这里成员i比成员j先声明，因此正确的顺序是先用j对i进行初始化，然后用2对j进行初始化。由于在对i进行初始化时j尚未被初始化，j的值为随机值，故i的值也为随机值；然后用2对j进行初始化，j的值为2。

【答案】

i为随机值，j为 2。在VisualC++ 6.0下输出为：

```
1    -858993460
2    2
```

面试题8 看代码写结果——静态成员变量的使用

考点：静态成员变量的理解和使用

出现频率：★★★★

```
1   #include <iostream>
2   using namespace std;
3
4   class Myclass
5   {
6   public:
7           Myclass(int a, int b, int c);
8           void GetNumber();
9           void GetSum();
10   private:
11       int A;
12       int B;
13       int C;
14       int Num;
15       static int Sum;
16   };
17
18   int Myclass::Sum = 0;
19
20   Myclass::Myclass(int a, int b, int c)
```

```
21  {
22      A = a;
23      B = b;
24      C = c;
25      Num = A+B+C;
26      Sum = A+B+C;
27  }
28
29  void Myclass::GetNumber()
30  {
31      cout << "Number = " << Num << endl;
32  }
33
34  void Myclass::GetSum()
35  {
36      cout << "Sum = " << Sum << endl;
37  }
38
39  void main()
40  {
41      Myclass M(3, 7, 10),N(14, 9, 11);
42
43      M.GetNumber();
44      N.GetNumber();
45      M.GetSum();
46      N.GetSum();
47  }
```

【解析】

本题考查的是静态成员与非静态成员的区别。静态成员被当作该类类型的全局变量。对于非静态成员，每个类对象都有自己的复制品，而静态成员对每个类的类型只有一个复制品。静态成员只有一份，由该类类型的所有对象共享访问。

Myclass类有GetNumber()和GetSum()两种方法，它们分别输出成员变量Num和Sum的值。main()函数中定义了两个 Myclass 的对象，并调用它们的 GetNumber()和 GetSum()方法。

Num成员为普通类型，它为Myclass类的对象所有。因此两次打印出来的值不一样。

Sum成员为静态类型，它为Myclass类所有，被Myclass类的所有对象所共享。因此，两次打印出来的值是相同的。

【答案】

程序输出结果为：

```
1    Number = 20
2    Number = 34
3    Sum = 34
4    Sum = 34
```

面试题9 与全局对象相比，使用静态数据成员有什么优势

考点：对静态成员变量的理解

出现频率：★★★★

【答案】

主要有以下两种优势。

静态数据成员没有进入程序的全局名字空间，因此不存在程序中其他全局名字冲突的可能性。

使用静态数据成员可以隐藏信息。因为静态成员可以是 private 成员，而全局对象不能。

面试题10 有哪几种情况只能用intialization list，而不能用assignment

考点：初始化列表和赋值的区别

出现频率：★★★

【解析】

无论是在构造函数初始化列表中初始化成员，还是在构造函数体中对它们赋值，最终结果都是相同的。不同之处在于，使用构造函数初始化列表初始化数据成员，没有定义初始化列表的构造函数在构造函数体中对数据成员赋值。

对于const和reference类型成员变量，它们只能够被初始化而不能做赋值操作，因此只能用初始化列表。

还有一种情况就是，类的构造函数需要调用其基类的构造函数的时候。请看下面的代码。

```
1    #include <iostream>
2    using namespace std;
3
4    class A                    //A是父类
5    {
6    private:
7        int a;                 //private成员
8    public:
9            A() {}
10       A(int x):a(x) {}       //带参数的构造函数对a初始化
11       void printA()          //打印a的值
```

```
12      {
13          cout << "a = " << a << endl;
14      }
15  };
16
17  class B : public A //B是子类
18  {
19  private:
20      int b;
21  public:
22      B(int x, int y) : A(x)    //需要初始化b以及父类的a
23      {
24          //a = x;        //a为private，无法在子类中被访问，编译错误
25          //A(x);            //调用方式错误，编译错误
26          b = y;
27      }
28      void printB() //打印b的值
29      {
30          cout << "b = " << b << endl;
31      }
32  };
33
34  int main()
35  {
36      B b(2,3);
37
38      b.printA();            //调用子类的printA()
```

```
39        b.printB();            //调用自己的printB()
40
41        return 0;
42    }
```

从上面的程序可以看到，如果在子类的构造函数中需要初始化父类的private成员，直接对其赋值是不行的（代码第 24 行），只有调用父类的构造函数才能完成对它的初始化。但在函数体内调用父类的构造函数也是不合法的（代码第25行），只有采取22行中的初始化列表调用子类构造函数的方式。程序的执行结果如下。

```
1    a = 2
2    b = 3
```

【答案】

当类中含有const、reference成员变量和基类的构造函数时都需要初始化列表。

面试题11 静态成员的错误使用

下面的代码有错误，找出来并说明原因。

考点：静态成员与非静态成员的理解

出现频率：★★★★

```
1    #include <iostream>
2    using namespace std;
3
4    class test
5    {
6    public:
7        static int i;
8        int j;
9        test(int a) : i(1), j(a) {}
10         void func1();
11         static void func2();
12    };
13
14    void test::func1() { cout << i << ", " << j << endl; }
15
16    void test::func2() { cout << i << ", " << j << endl; }
17
18    int main()
19    {
20         test t(2);
```

```
21        t.func1();
22        t.func2();
23        return 0;
24    }
```

【解析】

这个程序会出现如下两个错误。

（1）代码第9行，不能初始化i。

（2）代码第16行，在静态成员函数中非法引用了数据成员test::j。

第1个错误是关于静态成员变量i的初始化问题。为了与非静态成员变量相区别，i不能在类内部被初始化。可以把i放在类定义外面初始化（例如代码第13行）。

第2个错误是关于静态成员函数访问非静态成员的错误。要知道，静态成员函数和静态成员变量一样，不属于类的对象，因此它不含this指针，也就无法调用类的非静态成员。

下面是正确的代码。

```
1    #include <iostream>
2    using namespace std;
3
4    class test
5    {
6    public:
7            static int i;
8            int j;
9            test(int a) : j(a) {}
10        void func1();
11        static void func2();
12    };
```

13　int test::i = 1;

14　void test::func1() { cout << i << ", " << j << endl; }

15

16　void test::func2() { cout << i << /*", " << j <<*/ endl; } //注释对j的调用

17

18　int main()

19　{

20　　test t(2);

21　　t.func1();

22　　t.func2();

23　　return 0;

24　}

程序执行结果为：

1　1, 2

2　1

面试题12 对静态数据成员的正确描述

考点：对静态数据成员的理解和使用

出现频率：★★★

下面对静态数据成员的描述中，正确的是。

A．静态数据成员可以在类体内进行初始化

B．静态数据成员不可以被类的对象调用

C．静态数据成员不能受private控制符的作用

D．静态数据成员可以直接用类名调用

【解析】

A错误。静态数据成员必须在类外面初始化，以示与普通数据成员的区别。

B错误。

C正确。

D正确。

【答案】

C D

面试题13 main函数执行前还会执行什么代码?

考点：对构造函数调用期的理解

出现频率：★★★

【解析】

请看下面的程序代码。

```
1    #include <iostream>
2    using namespace std;
3
4    class Test
5    {
6    public:
7            Test()          //构造函数
8            {
9                cout << "constructor of Test" << endl;
10       }
11   };
12
13   Test a;                 //全局变量
14
15   int main()
16   {
17       cout << "main() start" << endl;
18       Test b;             //局部变量
```

```
19        return 0;
20    }
```

程序输出结果：

```
1    constructor of Test
2    main() start
3    constructor of Test
```

显然，这里的执行顺序为：首先进行全局对象a的构造，然后进入main函数中，再进行局部对象b的构造。

【答案】

全局对象的构造函数会在main函数之前执行。

面试题14 C++中的空类默认会产生哪些类成员函数

考点：编译器对C++类的默认处理

出现频率：★★★★★

【解析】

对于一个C++的空类，比如Empty：

```
1    class Empty
2    {
3    };
```

虽然 Empty 类定义中没有任何成员，但为了进行一些默认的操作，编译器会加入以下一些成员函数，这些成员函数使得类的对象拥有一些通用的功能。

默认构造函数和复制构造函数。它们被用于类的对象的构造过程。

析构函数。它被用于类的对象的析构过程。

赋值函数。它被用于同类的对象间的赋值过程。

取值运算。当对类的对象进行取地址（&）时，此函数被调用。

即虽然程序员没有定义类的任何成员，但是编译器也会插入一些函数，完整的 Empty类定义如下。

```
1    class Empty
2    {
3    public:
4    Empty();                        //缺省构造函数
5    Empty( const Empty& );              //复制构造函数
6    ~Empty();                        //析构函数
```

```
7    Empty& operator=( const Empty& );    //赋值运算符
8    Empty* operator&(); //取址运算符
9    const Empty* operator&() const;        //取址运算符 const
10    };
```

【答案】

C++的空类中，默认会产生默认构造函数、复制构造函数、析构函数、赋值函数以及取值运算。

面试题15 构造函数和析构函数是否可以被重载

考点：对构造函数和析构函数的理解

出现频率：★★★★

【答案】

构造函数可以被重载，因为构造函数可以有多个，且可以带参数。

析构函数不可以被重载。因为析构函数只能有一个，且不能带参数。

面试题16 关于重载构造函数的调用

考点：重载构造函数的调用

出现频率：★★★★

```
1   class Test
2   {
3   public:
4           Test() {}
5           Test(char *Name, int len = 0) { }
6           Test(char *Name) { }
7   };
8   int main()
9   {
10      Test obj("Hello");
11      return 0;
12   }
```

下面对程序执行结果的描述中，正确的是（　　）。

A．将会产生运行时错误

B．将会产生编译错误

C．将会执行成功

D．以上说法都不正确

【解析】

Test类定义了两个构造函数。当编译到代码第10行时，由于构造函数的模糊语义，编译器无法决定调用哪一个构造函数，因此会产生编译错误。

另外，如果把第10行注释掉，编译器将不会产生错误。因为C++编译器认为潜在的二义性不是一种错误。

【答案】

B

面试题17 构造函数的使用

考点：构造函数的使用

出现频率：★★★★

以下代码中的输出语句输出0吗？为什么?

```
1    #include <iostream>
2    using namespace std;
3
4    struct CLS
5    {
6            int m_i;
7            CLS( int i ) : m_i(i) {}
8            CLS()
9            {
10             CLS(0);
11         }
12   };
13   int main()
14   {
15       CLS obj;
16       cout << obj.m_i << endl;
17       return 0;
18   }
```

【解析】

在代码第10行，不带参数的构造函数直接调用了带参数的构造函

数。这种调用往往被很多人误解，以为可以通过构造函数的重载和相互调用实现一些类似默认参数的功能，其实是不行的，而且往往会有副作用。下面加几条打印对象地址的语句到原来的程序中。

```
1    #include <iostream>
2    using namespace std;
3
4    struct CLS
5    {
6            int m_i;
7            CLS( int i ) : m_i(i)
8            {
9                cout << "CLS(): this = " << this << endl;
10    }
11       CLS()
12       {
13           CLS(0);
14           cout << "CLS(int): this = " << this << endl;
15       }
16
17    };
18
19    int main()
20    {
21       CLS obj;
22       cout << "&obj = " << &obj << endl;
23       cout << obj.m_i << endl;
24       return 0;
```

```
25    }
```

程序执行结果如下。

```
1    CLS(): this = 0012FF20
2    CLS(int): this = 0012FF7C
3    &obj = 0012FF7C
4    -858993460
```

可以看到，在带参数的构造函数里打印出来的对象地址和对象obj的地址不一致。实际上，代码第13行的调用只是在栈上生成了一个临时对象，对于自己本身毫无影响。还可以发现，构造函数的互相调用引起的后果不是死循环，而是栈溢出。

【答案】

输出不为0，是个随机数。

原因是构造函数内调用构造函数只是在栈上生成了一个临时对象，对于自己本身毫无影响。

面试题18 构造函数explicit与普通构造函数的区别

考点：explicit构造函数的作用

出现频率：★★★

【解析】

explicit构造函数是用来防止隐式转换的。请看下面的代码。

```
1    class Test1
2    {
3    public:
4            Test1(int n) { num = n; }        //普通构造函数
5    private:
6            int num;
7    };
8
9    class Test2
10   {
11   public:
12       explicit Test2(int n) { num = n; } //explicit(显式)构造函数
13   private:
14       int num;
15   };
16
17   int main()
18   {
```

```
19      Test1 t1 = 12;          //隐式调用其构造函数,成功
20      Test2 t2 = 12;          //编译错误,不能隐式调用其构造函数
21      Test2 t3(12);          //显示调用成功
22      return 0;
23  }
```

Test1的构造函数带一个int型的参数，代码第19行会隐式转换成调用Test1的这个构造函数。而Test2的构造函数被声明为explicit（显式），这表示不能通过隐式转换来调用这个构造函数，因此代码第20行会出现编译错误。

【答案】

普通构造函数能够被隐式调用，而explicit构造函数只能被显示调用。

面试题19 explicit构造函数的作用

考点：explicit用于构造函数的作用

出现频率：★★★

下面的程序f()被调用时，输出是什么？

```
1   #include <iostream>
2   #include <string>
3   using namespace std;
4
5   class Number
6   {
7   public:
8           string type;
9           Number(): type("void") { }
10      explicit Number(short) : type("short") { }
11      Number(int) : type("int") { }
12   };
13
14   void Show(const Number& n) { cout << n.type; }
15
16   void main()
17   {
18      short s = 42;
19      Show(s);
20   }
```

A．void

B．short

C．int

D．None of the above

【解析】

Show()函数的参数类型是Number类对象的引用，代码第19行调用Show(s)时采取了以下所示的步骤。

（1）Show(s)中的s为short类型，其值为42，因此首先检查参数为short的构造函数能否被隐式转换。由于代码第 10 行的构造函数被声明为显式调用（explicit），因此不能隐式转换。于是进行下一步。

（2）42自动转换为int类型。

（3）检查参数为int的构造函数能否被隐式转换。由于代码第11行参数为int的构造函数没有被声明为显式调用，因此调用此构造函数构造出一个临时对象。

（4）打印上一步临时对象的type成员，即"int"。

【答案】

C

面试题20 C++中虚析构函数的作用是什么

考点：对虚构造函数的理解

出现频率：★★★

【解析】

大家知道，析构函数是为了在对象不被使用之后释放它的资源，虚函数是为了实现多态。那么，把析构函数声明为virtual有什么作用呢？请看下面的代码。

```
1   #include <iostream>
2   using namespace std;
3
4   class Base
5   {
6   public:
7           Base() {};          //Base的构造函数
8           ~Base()             //Base的析构函数
9       {
10              cout << "Output from the destructor of class Base!" <<
endl;
11      };
12      virtual void DoSomething()
13      {
14              cout << "Do something in class Base!" << endl;
15      };
16  };
```

```
17
18  class Derived : public Base
19  {
20  public:
21      Derived() {};       //Derived的构造函数
22      ~Derived()          //Derived的析构函数
23      {
24          cout << "Output from the destructor of class Derived!" <<
endl;
25      };
26      void DoSomething()
27      {
28          cout << "Do something in class Derived!" << endl;
29      };
30  };
31
32  int main()
33  {
34      Derived *pTest1 = new Derived();    //Derived类的指针
35      pTest1->DoSomething();
36      delete pTest1;
37
38      cout << endl;
39
40      Base *pTest2 = new Derived();    //Base类的指针
41      pTest2->DoSomething();
42      delete pTest2;
```

```
43
44      return 0;
45  }
```

先看程序输出结果：

```
1   Do something in class Derived!
2   Output from the destructor of class Derived!
3   Output from the destructor of class Base!
4
5   Do something in class Derived!
6   Output from the destructor of class Base!
```

代码第36行可以正常释放pTest1的资源，而代码第42行没有正常释放pTest2的资源，因为从结果看，Derived类的析构函数并没有被调用。通常情况下，类的析构函数里面都是释放内存资源，而析构函数不被调用的话就会造成内存泄漏。原因是指针pTest2是Base类型的指针，释放pTest2时只进行Base类的析构函数。在代码第8行前面加上virtual关键字后的运行结果如下。

```
1   Do something in class Derived!
2   Output from the destructor of class Derived!
3   Output from the destructor of class Base!
4
5   Do something in class Derived!
6   Output from the destructor of class Derived!
7   Output from the destructor of class Base!
```

此时释放指针pTest2时，由于Base的析构函数是virtual的，就会先找到并执行Derived类的析构函数，然后执行Base类的析构函数，资源正常释放，避免了内存泄漏。

因此，只有当一个类被用来作为基类的时候，才会把析构函数写成

虚函数。

面试题21 看代码写结果——析构函数的执行顺序

考点：构造函数的执行顺序与构造函数相反

出现频率：★★★★

```
1   #include<iostream.h>
2   class A
3   {
4   private:
5           int a;
6
7   public:
8           A(int aa) { a = aa; };
9           ~A() { cout<<"Destructor A!"<<a<<endl; };
10   };
11
12   class B:public A
13   {
14   private:
15       int b;
16
17   public:
18       B( int aa = 0, int bb = 0 ):A(aa) { b = bb; };
19       ~B(){ cout<<"Destructor B!"<<b<<endl; };
20   };
```

```
21
22  void main()
23  {
24      B obj1(5), obj2(6, 7);
25      return;
26  };
```

【解析】

本题考查的是析构函数的执行顺序。析构函数的执行顺序与构造函数的执行顺序相反。

main()函数中定义了两个类B的对象，它们的基类是A。由于这两个对象都是栈中分配的，当main()函数退出时会发生析构，又因为obj1比obj2先声明，所以obj2先析构。它们析构的顺序是首先执行B的析构函数，然后执行A的析构函数。

【答案】

程序输出如下。

```
1   Destructor B!7
2   Destructor A!6
3   Destructor B!0
4   Destructor A!5
```

面试题22 复制构造函数是什么？什么是深复制和浅复制

复制构造函数是什么？什么情况下会用到它？什么是深复制和浅复制？

考点：对复制构造函数的理解

出现频率：★★★

【解析】

先来说明什么是复制构造函数，以及它被调用的场合。

复制构造函数是一种特殊的构造函数，它由编译器调用来完成一些基于同一类的其他对象的构件及初始化。

如果在类中没有显式地声明一个复制构造函数，那么，编译器会私下里制定一个函数来进行对象之间的位复制（bitwise copy）。这个隐含的复制构造函数简单地关联了所有的类成员。

在C++中，下面是3种对象需要复制的情况。因此，复制构造函数将会被调用。

（1）一个对象以值传递的方式传入函数体。

（2）一个对象以值传递的方式从函数返回。

（3）一个对象需要通过另外一个对象进行初始化。

下面的程序代码说明了上述三种情况。

```
1    #include <iostream>
2    using namespace std;
3
4    class Test
5    {
```

```
6   public:
7          int a;
8          Test(int x)
9          {
10           a = x;
11        }
12        Test(Test &test)           //复制构造函数
13        {
14           cout << "copy constructor" << endl;
15           a = test.a;
16        }
17    };
18
19    void fun1(Test test)           //(1)值传递传入函数体
20    {
21        cout << "fun1()..." << endl;
22    }
23
24    Test fun2()                    //(2)值传递从函数体返回
25    {
26        Test t(2);
27        cout << "fun2()..." << endl;
28        return t;
29    }
30
31    int main()
32    {
```

```
33      Test t1(1);
34      Test t2 = t1;            //(3)用t1对t2做初始化
35      cout << "before fun1()..." << endl;
36      fun1(t1);
37
38      Test t3 = fun2();
39      cout << "after fun2()..." << endl;
40      return 0;
41  }
```

程序执行结果如下。

```
1  copy constructor
2  before fun1()…
3  copy constructor
4  fun1()…
5  fun2()…
6  copy constructor
7  copy constructor
8  after fun2()…
```

fun1()、fun2()以及代码第34行分别对应了上面3种调用复制构造函数的情况。

接下来说明深复制与浅复制。

既然系统会自动提供一个默认的复制构造函数来处理复制，那么，为什么要去自定义复制构造函数呢？下面的程序代码说明了这个问题。

```
1  #include <iostream>
2  using namespace std;
3
4  class Test
```

```
5   {
6   public:
7         char *buf;
8         Test(void)                          //不带参数的构造函数
9         {
10          buf = NULL;
11       }
12       Test(const char* str)                //带参数的构造函数
13       {
14          buf = new char[strlen(str) + 1];  //分配堆内存
15          strcpy(buf, str);                 //复制字符串
16       }
17       ~Test()
18       {
19          if (buf != NULL)
20          {
21                delete buf;                 //释放buf指向的堆内存
22                buf = NULL;
23          }
24       }
25   };
26
27   int main()
28   {
29       Test t1("hello");
30       Test t2 = t1;                        //调用默认的复制构造函数
31
```

```
32        cout << "(t1.buf == t2.buf) ? " <<
33            (t1.buf == t2.buf ? "yes": "no") << endl;
34
35        return 0;
36    }
```

这里 Test 类的 buf 成员是一个字符指针，在带参数的构造函数中为之分配了一块堆内存来存放字符串，然后在析构函数中又将堆内存释放。在main()函数（代码第30行）使用了对象复制，因此会调用默认的复制构造函数。程序的执行结果如下。

```
1    (t1.buf == t2.buf) ? yes
2    程序崩溃
```

这里程序崩溃发生在main()函数退出对象析构的时候。由前两行的打印结果可以看出，默认复制构造函数只是简单地把两个对象的指针做赋值运算，它们指向的是同一个地址。当产生两次析构，释放同一块堆内存时发生崩溃。可以在 Test 类里通过添加一个自定义的复制构造函数解决两次析构的问题。

```
1    Test(Test &test)
2    {
3        buf = new char[strlen(test.buf) + 1];
4        strcpy(buf, test.buf);
5    }
```

程序执行结果如下。

```
1    (t1.buf == t2.buf) ? no
```

由于此时buf又分配了一块堆内存来保存字符串，t1的buf和t2的buf分别指向不同的堆内存，析构时就不会发生程序崩溃。

总结：如果复制的对象中引用了某个外部的内容（例如分配在堆上的数据），那么在复制这个对象的时候，让新旧两个对象指向同一个外

部的内容，就是浅复制；如果在复制这个对象的时候为新对象制作了外部对象的独立复制，那么就是深复制。

【答案】

复制构造函数是一种特殊的构造函数，它由编译器调用来完成一些基于同一类的其他对象的构件及初始化。

浅复制是指让新旧两个对象指向同一个外部的内容，而深复制是指为新对象制作了外部对象的独立复制。

面试题23 编译器与默认的copy constructor

什么时候编译器会生成默认的copy constructor呢？如果已经写了一个构造函数，编译器还会生成copy constructor吗？

考点：对复制构造函数的理解

出现频率：★★★

【答案】

如果用户没有自定义复制构造函数，并且在代码中用到了复制构造函数，那么编译器就会生成默认的复制构造函数；但如果用户定义了复制构造函数，那么编译器就不会再生成复制构造函数。

如果用户定义了一个构造函数，且不是复制构造函数，而此时在代码中用到了复制构造函数，那么编译器也还会生成默认的复制构造函数；如果没有使用，则编译器就不会生成默认的复制构造函数。

面试题24 写一个继承类的复制函数

考点：对继承类的复制构造函数的理解

出现频率：★★★

【解析】

当然，如果基类中没有私有成员，即所有成员都能被派生类访问，则派生类的复制构造函数可以很容易写。但如果基类有私有成员，并且这些私有成员必须在调用派生类的复制构造函数时被初始化，在这种情况下又该怎么做呢？

编写继承类的复制函数有一个原则：使用基类的复制构造函数。这个原则其实就是解决上述问题的方案。请看下面的程序代码。

```
1    #include <iostream>
2    using namespace std;
3
4    class Base
5    {
6    public:
7        Base():i(0) {cout << "Base()" << endl;}            //默认普通构造函数
8        Base(int n):i(n) {cout << "Base(int)" << endl;}      //普通构造函数
9        Base(const Base &b) :i(b.i) //复制构造函数
10       {
11          cout << "Base(Base&)" << endl;
12       }
```

```
13  private:
14      int i;   //私有成员
15  };
16
17  class Derived : public Base
18  {
19  public:
20      Derived():Base(0), j(0) {cout << "Derived()" << endl;}   //默认
普通构造函数
21      Derived(int m, int n):Base(m), j(n) {cout << "Derived(int)" <<
endl;}   //普通
    构造函数
22      Derived(Derived &obj) : Base(obj), j(obj.j)          //Derived类的
复制构造函数
23      {                                                   //调用了Base的复制构造函
数
24          cout << "Derived(Derived&)" << endl;
25      }
26  private:
27      int j;
28  };
29
30  int main()
31  {
32      Base b(1);
33      Derived obj(2, 3);
34      cout << "——————————————————————————" << endl;
```

35　　Derived d(obj);　　　　　　　　　　//调用Derived的复制构造函数

36　　cout << "——————————————" << endl;

37　　return 0;

38　}

Derived类继承自Base类，因此在Derived类内不能使用obj.i或Base::i的方式访问Base的私有成员i。很明显，其复制构造函数只有使用Base(obj)（代码第22行）的方式调用其基类的复制构造函数来给基类的私有成员i初始化。

面试题25 复制构造函数与赋值函数有什么区别

考点：构造函数与赋值函数的区别

出现频率：★★★

【解析】

有3个方面的区别：

（1）复制构造是一个对象来初始化一块内存区域，这块内存就是新对象的内存区。例如

```
1    class A;
2    A a;
3    A b=a;    //复制构造函数调用
4
5    A b(a);    //复制构造函数调用
```

而赋值函数是对于一个已经被初始化的对象来进行 operator=操作。例如

```
1    class A;
2    A a;
3    A b;
4    b = a;    //赋值函数调用
```

（2）一般来说是在数据成员包含指针对象的时候，应付两种不同的处理需求：一种是复制指针对象，一种是引用指针对象。复制构造函数在大多数情况下是复制，赋值函数则是引用对象。

（3）实现不一样。复制构造函数首先是一个构造函数，它调用的时候是通过参数传进来的那个对象来初始化产生一个对象。赋值函数则

是把一个对象赋值给一个原有的对象，所以，如果原来的对象中有内存分配，要先把内存释放掉，而且还要检查一下两个对象是不是同一个对象，如果是的话，就不做任何操作。

面试题26 编写类String的构造函数、析构函数和赋值函数

考点：构造函数、析构函数和赋值函数的编写方法

出现频率：★★★★★

已知类String的原型为：

```
1   class String
2   {
3   public:
4           String(const char *str = NULL);   //普通构造函数
5           String(const String &other);      //复制构造函数
6           ~ String(void);                   //析构函数
7           String & operator =(const String &other); //赋值函数
8   private:
9           char *m_String;                   //私有成员，保存字符串
10  };
```

【解析】

程序代码如下。

```
1   #include <iostream>
2   using namespace std;
3
4   class String
5   {
6   public:
7           String(const char *str = NULL);         //普通构造函数
```

```
8          String(const String &other);               //复制构造函数
9          ~ String(void);                            //析构函数
10       String & operator =(const String &other); //赋值函数
11    private:
12       char *m_String;                             //私有成员，保存字符串
13    };
14
15    String::~String(void)
16    {
17       cout << "Destructing"<< endl;
18       if (m_String != NULL)                   //如果m_String不为
NULL，释放堆内存
19       {
20             delete [] m_String;
21             m_String = NULL;                  //释放后置为NULL
22       }
23    }
24
25    String::String(const char *str)
26    {
27       cout << "Construcing" << endl;
28       if(str == NULL)                         //如果str为NULL，存空字符
串""
29       {
30          m_String = new char[1];           //分配一个字节
31             *m_String = '\0';              //将之赋值为字符串结束符
32       }
```

```
33      else
34      {
35          m_String = new char[strlen(str) + 1]; //分配空间容纳str内容
36          strcpy(m_String, str);          //复制str到私有成员
37      }
38  }
39
40  String::String(const String &other)
41  {
42      cout << "Constructing Copy" << endl;
43      m_String = new char[strlen(other.m_String) + 1];    //分配空间
容纳str内容
44      strcpy(m_String, other.m_String);                //复制str到私有成
员
45  }
46
47  String & String::operator = (const String &other)
48  {
49      cout << "Operate = Function" << endl;
50      if(this == &other)                          //如果对象与other是同一
个对象
51      {                                           //直接返回本身
52          return *this;
53      }
54      delete [] m_String;                         //释放堆内存
55      m_String = new char[strlen(other.m_String)+1];
56      strcpy(m_String, other.m_String);
```

```
57
58        return *this;
59    }
60
61    int main()
62    {
63        String a("hello");                    //调用普通构造函数
64        String b("world");                    //调用普通构造函数
65        String c(a);                          //调用复制构造函数
66        c = b;                                //调用赋值函数
67
68        return 0;
69    }
```

（1）普通构造函数：这里判断了传入的参数是否为NULL。如果是NULL，初始化一个字节的空字符串（包括结束符'\0'）；如果不是，分配足够大小长度的堆内存来保存字符串。

（2）复制构造函数：只是分配足够小长度的堆内存来保存字符串。

（3）析构函数：如果类私有成员m_String不为NULL，释放m_String指向的堆内存，并且为了避免产生野指针，将m_String赋为NULL。

（4）赋值函数：首先判断当前对象与引用传递对象是否是同一个对象，如果是，不做操作，直接返回；否则，先释放当前对象的堆内存，然后分配足够大小长度的堆内存复制字符串。

程序的执行结果如下。

```
1    Construcing
2    Construcing
```

```
3    Construcing Copy
4    Operate = Function
5    Destructing
6    Destructing
7    Destructing
```

这里代码第 63~66 行会发生构造函数以及赋值函数的调用，而析构函数的调用发生在main()函数退出时。

面试题27 了解C++类各成员函数的关系

写出下面代码的输出结果。

考点：构造函数、析构函数和赋值函数的关系

出现频率：★★★★

```
1    #include <iostream.h>
2
3    class A
4    {
5    private:
6            int num;
7    public:
8            A()
9            {
10             cout<<"Default constructor"<< endl;
11         }
12         ~A()
13         {
14             cout << "Desconstructor" << endl;
15             cout << num << endl;
16         }
17         A(const A &a)
18         {
19             cout << "Copy constructor" << endl;
20         }
```

```
21        void operator = (const A &a) {
22            cout << "Overload operator" << endl;
23        }
24        void SetNum(int n) {
25            num = n;
26        }
27    };
28
29    void main()
30    {
31        A a1;
32        A a2(a1);
33        A a3=a1;
34        A &a4=a1;
35        a1.SetNum(1);
36        a2.SetNum(2);
37        a3.SetNum(3);
38        a4.SetNum(4);
39    }
```

【解析】

代码第31行，定义了一个对象a1，调用的是默认的构造函数。

代码第32行，用a1初始化一个对象a2，调用的是复制构造函数。

代码第 33 行，同上。注意，这里不是调用赋值函数，这里属于对象 a3 的初始化，而不是赋值。若要调用赋值，必须为如下形式。

```
1    A a3;
2    a3 = a1;
```

代码第34行，定义a4为a1的一个引用，不调用构造函数或赋值函

数。

代码第35~38行，调用各个对象的SetNum()成员函数为私有成员num赋值。这里注意，由于a4为a1的引用，因此a4.SetNum()实际上和a1.SetNum()等同。

当 main()函数退出时，对象析构顺序与调用构造函数顺序相反，依次为 a3、a2和a1。

【答案】

程序执行结果：

```
1    Default constructor
2    Copy constructor
3    Copy constructor
4    Desconstructor
5    3
6    Desconstructor
7    2
8    Desconstructor
9    4
```

面试题28 C++类的临时对象

考点：构造函数、析构函数和赋值函数的编写方法

出现频率：★★★★

已知class B 以及Play()函数定义如下。

```
1    #include <iostream.h>
2
3    class B
4    {
5    public:
6            B()
7            {
8                cout<<"default constructor"<<endl;
9            }
10
11         ~B()
12         {
13             cout<<"destructed"<<endl;
14         }
15
16         B(int i) : data(i)           //初始化私有成员data
17         {
18             cout<<"constructed by parameter " << data <<endl;
19         }
20
```

```
21   private:
22       int data;
23   };
24
25   B Play( B b)
26   {
27       return b ;
28   }
```

分析下面两个main()函数的输出。

第1个main()函数：

```
1   int main(int argc, char* argv[])
2   {
3           B t1 = Play(5);
4           B t2 = Play(t1);
5
6           return 0;
7   }
```

第 2个main()函数：

```
1   int main(int argc, char* argv[])
2   {
3           B t1 = Play(5);
4           B t2 = Play(10);
5
6           return 0;
7   }
```

【解析】

这里调用Play()函数时，有两种参数类型的传递方式。

如果传递的参数是整型数，那么在其函数栈中首先会调用带参数的构造函数，产生一个临时对象，然后返回前（在return代码执行时）调用类的复制构造函数，生成临时对象（这样函数返回后主函数中的对象就被初始化了），最后这个临时对象会在函数返回时（在return代码执行后）析构。

如果传递的参数是 B 类的对象，那么只有第一步与上面的不同，就是其函数栈中会首先调用复制构造函数产生一个临时对象，其余步骤完全相同。

可以看出，两种情况的区别是采用了不同的方法生成临时对象（一个是调用带参数的构造函数，另一个是调用复制构造函数）。

在第一个 main()函数中，对象 t1 使用了传入整型数的方式调用Play()函数，而对象 t2使用了传入B的对象的方式调用Play()函数。在第二个main()函数中，对象t1和t2都使用了传入整型数的方式调用Play()函数。第一个main()函数下的执行结果为：

1　constructed by parameter 5 （调用带参数的构造函数，在fun内生成临时对象）

2　destructed　　　（5传入fun时生成的临时对象析构）

3　destructed　　　（t1传入fun时产生的返回的临时对象析构）

4　destructed　　　（t2析构）

5　destructed　　　（t1析构）

第二个main()函数下的执行结果为：

1　constructed by parameter 5 （调用带参数的构造函数，在fun内产生临时对象）

2　destructed　　　（5传入fun时生成的临时对象析构）

3　constructed by parameter 10（调用带参数的构造函数，在fun内产生临时对象）

4　destructed　　　（10传入fun时生成的临时对象析构）

5　destructed　　　（t2析构）

6　destructed　　　（t1析构）

为了更加详细地说明结果，在B类中加入一个自定义的复制构造函数：

```
1	B(B &b)
2	{
3	    cout << "copy constructor" << endl;
4	    data = b.data;
5	}
```

第一个main()函数下的执行结果为：

1　constructed by parameter 5 （调用带参数的构造函数，在fun内产生临时对象）

2　copy constructor　　（调用复制构造函数，把临时对象复制到t1）

3　destructed　　　（fun内的临时对象析构）

4　copy constructor　　（调用复制构造函数，在fun内产生临时对象）

5　copy constructor　　（调用复制构造函数，把临时对象复制到t2）

6　destructed　　　（fun内的临时对象析构）

7　destructed　　　（t2析构）

8　destructed　　　（t1析构）

第二个main()函数下的执行结果为：

1　constructed by parameter 5 （调用带参数的构造函数，在fun内产生临时对象）

2　copy constructor　　（调用复制构造函数，把临时对象复制到t1）

3　destructed　　　（fun内的临时对象析构）

4　constructed by parameter 10 （调用带参数的构造函数，在fun内产生临时对象）

5　copy constructor　　（调用复制构造函数，把临时对象复制到t2）

6　destructed　　　（fun内的临时对象析构）

7　destructed　　　（t2析构）

8　destructed　　　（t1析构）

此时，两个main()函数的输出结果只有第4行不一样，也就是由于传入不同类型参数（整型与对象类型），这是由采取了不同的方式生成临时对象而导致的。

面试题29 复制构造函数和析构函数

考点：对复制构造函数和析构函数的理解

出现频率：★★★★

```
1   #include  <iostream.h>
2   class  A
3   {
4   public:
5         A()
6         {
7            cout << "This is A Construction" << endl;
8         }
9         virtual ~A()
10       {
11          cout << "This is A destruction" << endl;
12       }
13   };
14
15   A fun()
16   {
17      A a;
18      return a;
19   }
20
21   void main()
```

```
22  {
23      {
24          A a;
25          a = fun();
26      }
27  }
```

已知程序输出为：

```
1  This is A Construction
2  This is A Construction
3  This is A destruction
4  This is A destruction
5  This is A destruction
```

不是说构造函数和析构函数是成对的吗？为什么少了一个构造函数呢？

【解析】

构造函数和析构函数确实是成对的。构造函数除了普通构造函数之外，还包括复制构造函数。

上面的程序中一共构造了3个对象，分别是main()函数中的a（代码第24行）、fun()函数中的a（代码第17行）以及fun返回时生成的临时对象（代码第18行）。前两个对象都是用普通构造函数构造的，而由 fun返回时生成的临时对象是由复制构造函数生成的。上面的程序中只是在普通构造函数中打印了信息。加入自定义复制构造函数和赋值函数，如下所示。

```
1  A(A &a)
2  {
3      cout << "This is A Copy Construction" << endl;
4  }
```

```
5    A& operator =(const A &a)
6    {
7        cout << "This is an assignment function" << endl;
8        return *this;
9    }
```

程序的执行结果如下。

```
1    This is A Construction
2    This is A Construction
3    This is A Copy Construction
4    This is A destruction
5    This is an assignment function
6    This is A destruction
7    This is A destruction
```

可以看出，此时的构造函数和析构函数都被执行了3次。另外，在main()函数中把fun()返回的临时对象赋给了对象a，此时会调用赋值函数。

【答案】

构造函数和析构函数确实是成对的，原程序中的fun返回时生成的临时对象是由复制构造函数生成的。这里没有在复制构造函数中输出信息（编译器生成默认的复制构造函数），所以看上去构造函数比析构函数少了一个。

面试题30 看代码写结果——C++静态成员和临时对象

考点：对C++静态成员和临时对象的理解

出现频率：★★★★

```
1    #include <iostream>
2    using namespace std ;
3
4    class human
5    {
6    public:
7        human()
8        {
9            human_num++;
10       }
11       static int human_num;
12       ~human()
13       {
14           human_num—
15           print();
16       }
17       void print()
18       {
19           cout<<"human nun is: "<<human_num<<endl;
20       }
```

```
21  };
22
23  int human::human_num = 0;
24
25  human f1(human x)
26  {
27      x.print();
28      return x;
29  }
30
31  int main(int argc, char* argv[])
32  {
33      human h1;
34      h1.print();
35      human h2    = f1(h1);
36      h2.print();
37
38      return 0;
39  }
```

【解析】

这个程序的 human 类有一个静态成员 human_num ， 每执行一次，普通构造函数human_num加1，每执行一次，析构函数human_num减1。注意，在f1()函数中会使用默认的复制构造函数，而默认的复制构造函数没有对human_num处理。

代码第34行，只构造了对象h1（调用普通构造函数），因此打印1。

代码第35行使用值传递参数的方式调用了f1()函数，这里分为3步：

（1）在f1()函数内首先会调用复制构造函数生成一个临时对象，因此代码第27行打印1。

（2）f1()函数内调用复制构造函数，给main的对象h2初始化（复制临时对象）。

（3）f1()函数返回后，临时对象发生析构，此时human的静态成员human_num为0，打印出0。

代码第36行打印的还是0。

main()函数结束时有h1和h2两个对象要发生析构，所以分别打印出-1和-2。

程序的意图其实很明显，就是静态成员用human_num记录类human的实例数。然而，由于默认的复制构造没有对静态成员操作，导致了执行结果的不正确。这里可以通过添加一个自定义的复制构造函数解决。

```
1    human(human &h)
2    {
3        human_num++;
4    }
```

此时 human_num就能起到应有的作用了。

【答案】

程序执行结果：

```
1    1
2    1
3    0
4    0
5    -1
6    -2
```

面试题31 什么是临时对象？临时对象在什么情况下产生

考点：对C++临时对象的理解

出现频率：★★★★

【解析】

当程序员之间进行交谈时，经常把仅仅需要一小段时间的变量称为临时变量。例如在下面的swap()函数里：

```
1    void swap(int &a, int &b)
2    {
3        int temp = a;
4        a = b;
5        b = temp;
6    }
```

通常称temp为临时变量。但是在C++里，temp根本不是临时变量。实际上，它只是一个函数的局部变量。

真正的临时对象是看不见的，它不会出现在程序代码中。大多数情况下，它会影响程序执行的效率，所以有时想避免临时对象的产生。它通常在以下两种情况下产生。

（1）参数按值传递。

（2）返回值按值传递。

参考下面的代码。

```
1    #include <iostream>
2    using namespace std ;
3
```

```
4   class Test
5   {
6   public:
7       Test() : num(0) { }             //默认构造函数
8       Test(int number) : num(number) { }   //带参数的构造函数
9       void print()                    //打印私有成员num
10      {
11          cout << "num = " << num << endl;
12      }
13      ~Test()                         //析构函数，打印this指针和私有成员
num
14      {
15          cout << "destructor: this = " << this << ", num = " << num <<
endl;
16      }
17  private:
18      int num;
19  };
20
21  void fun1(Test test)               //参数按值传递
22  {
23      test.print();
24  }
25
26  Test fun2()
27  {
28      Test t(3);
```

```
29      return t;                    //返回值按值传递
30  }
31
32  int main(int argc, char* argv[])
33  {
34      Test t1(1);
35
36      fun1(t1);                     //对象传入
37      fun1(2);                     //整型数2传入
38      t1 = fun2();
39
40      return 0;
41  }
```

程序中的 fun1()函数的参数是按值传递的，fun2()函数的返回值是按值传递的。在 Test类的析构函数中打印了this指针的值，下面是程序执行结果。

```
1   num = 1
2   destructor: this = 0012FF0C, num = 1 （fun1内的临时对象析构）
3   num = 2
4   destructor: this = 0012FF0C, num = 2 （fun1内的临时对象析构）
5   destructor: this = 0012FEF4, num = 3 （fun2内的局部变量析构）
6   destructor: this = 0012FF68, num = 3 （fun2返回的临时对象析构）
7   destructor: this = 0012FF70, num = 3 （main内的t1析构）
```

这里代码第36行和第37行使用了两种类型的参数传入fun1()函数。它们都会生成临时变量，不同点是要采用不同的方式：第36行的调用使用了复制构造函数创建临时变量，而第37行调用使用的则是带参数的构

造函数创建临时变量。

如何避免临时变量的产生呢？可以使用按引用传递代替按值传递。例如，把上面的fun1()函数改成如下形式。

```
1    void fun1(Test &test)
2    {
3            test.print();
4    }
```

这样，fun1()函数的参数就是一个已经存在的对象引用，此时整型值是不能传进来的。执行下面的主程序。

```
1    int main(int argc, char* argv[])
2    {
3            Test t1(1);
4            fun1(t1); //对象引用传入
5            return 0;
6    }
```

程序的执行结果如下。

```
1    num = 1
2    destructor: this = 0012FF70, num = 1（main内的t1析构）
```

可以看到，此时就不会产生临时对象了。

注意：引用必须有一个实在的、可引用的对象，否则引用是错误的。因此，在没有实在的、可引用的对象的时候，只有依赖于临时对象。

面试题32 为什么C语言不支持函数重载而C++能支持

什么是函数重载？为什么C语言不支持函数重载，而C++能支持函数重载？

考点：对函数重载以及C++和C语言之间的差异性的理解

出现频率：★★★★

【解析】

函数重载是用来描述同名函数具有相同或者相似的功能，但数据类型或者是参数不同的函数管理操作。例如，要进行两种不同数据类型的和的操作，在 C 语言里需要写两个不同名称的函数来进行区分。

```
1    int add1(int a, int b)
2    {
3            return a + b;
4    }
5
6    float add2(float a, float b)
7    {
8            return a + b;
9    }
```

上面的代码写得不太好，这两个具备相似操作的函数，却给它们取了两个不同的名字，这样做不便于管理。因此，C++为了方便程序员编写，引入了函数重载的概念。例如下面的代码。

```
1    #include <iostream>
2    using namespace std;
```

```
3
4   class Test
5   {
6   public:
7          int add(int x, int y)       //相加,传入参数以及返回值都是int
8          {
9              return x+y;
10        }
11        float add(float x, float y)    //相加,传入参数以及返回值都是
float
12        {
13              return x+y;
14        }
15  };
16
17  int add(int x, int y)           //相加,传入参数以及返回值都是int
18  {
19      return x+y;
20  }
21
22  float add(float x, float y)       //相加,传入参数以及返回值都是
float
23  {
24      return x+y;
25  }
26
27  int main(int argc, char* argv[])
```

```
28  {
29      int i = add(1, 2);
30      float f = add(1.1f, 2.2f);
31      Test test;
32      int i1 = test.add(3, 4);
33      float f1 = test.add(3.3f, 4.4f);
34
35      cout << "i = " << i << endl;
36      cout << "f = " << f << endl;
37      cout << "i1 = " << i1 << endl;
38      cout << "f1 = " << f1 << endl;
39
40      return 0;
41  }
```

上面的程序中使用了全局函数和类成员函数的重载，代码第 29~38 行是对它们的调用与测试。可以看到，在 C++中可以根据传入参数类型和返回类型来区分不同的重载函数。

C语言不支持函数重载，C++却支持，为什么呢？这是因为C++的重载函数经过编译器处理之后，两个函数的符号是不相同的。例如代码第17行的add函数，经过处理后变成了_int_add_int_int之类，而后者变成了_float_add_float_float之类。这样的名字包含了函数名、函数参数数量及返回类型信息，C++就是靠这种机制来实现函数重载的。

【答案】

函数重载是用来描述同名函数具有相同或者相似的功能，但数据类型或者是参数不同的函数管理操作。

函数名经过 C++编译器处理后包含了原函数名、函数参数数量及返回类型信息，而 C语言不会对函数名进行处理。

面试题33 判断题——函数重载的正确声明

考点：函数重载的正确声明

出现频率：★★★★

```
1  (A)  int calc(int,int);
2       int calc(const int,const int);
3
4  (B)  int get();
5       double get();
6
7  (C)  int *reset(int *);
8       double *reset(double *);
9
10  (D)  extern "C" int compute(int *,int);
11       extern "C" double compute(double *,double);
```

【解析】

A错误。第二个函数被视为重复声明，第二个声明中的const修饰词会被忽略。

B错误。第二个声明是错误的，因为单就函数的返回值而言，不足以区分两个函数的重载。

C正确。这是合法的声明，reset()函数被重载。

D 错误。第二个函数声明是错误的，因为在一组重载函数中，只能有一个函数被指定为extern "C"。

【答案】

C正确。

面试题34 重载和覆写有什么区别

考点：对重载和覆写之间的区别的理解

出现频率：★★★★

【解析】

重载（overriding）是指子类改写了父类的方法，覆写（overloading）是指同一个函数的不同版本之间参数不同。

重载是编写一个与已有函数同名但是参数表不同（参数数量或参数类型不同）的方法，它具有如下所示的特征。

（1）方法名必须相同。

（2）参数列表必须不相同，与参数列表的顺序无关。

（3）返回值类型可以不相同。

覆写是派生类重写基类的虚函数，它具有如下所示的特征。

（1）只有虚方法和抽象方法才能够被覆写。

（2）相同的函数名。

（3）相同的参数列表。

（4）相同的返回值类型。

重载是一种语法规则，由编译器在编译阶段完成，不属于面向对象的编程；而覆写是由运行阶段决定的，是面向对象编程的重要特征。

面试题35 编程题——MyString类的编写

考点：重载=和+运算符

出现频率：★★★

对于下面的类MyString，要求重载一些运算符后可以计算表达式a=b+c;。

其中，a、b、c都是类MyString的对象。请重载相应的运算符并编写程序测试。

```
1   class MyString
2   {
3   public:
4           MyString(char *s)
5           {
6               str = new char[strlen(s)+1];
7               strcpy(str,s);
8           }
9           ~MyString()
10         {
11             delete []str;
12         }
13   private:
14       char *str;
15   };
```

【解析】

为了实现a=b+c;这个表达式，需要重载两个运算符，一个是 '+' 运

算符，用于b+c，另一个是 '=' 运算符，用于对象a 的赋值。程序代码如下。

```
1    #include <iostream>
2    using namespace std;
3
4    class MyString
5    {
6    public:
7        MyString(char *s)            //参数为字符指针的构造函数
8        {
9        str = new char[strlen(s)+1];
10         strcpy(str,s);
11       }
12       ~MyString()                  //析构函数释放str堆内存
13       {
14         delete []str;
15       }
16       MyString & operator = (MyString &string)      //赋值函数，重
载 =
17       {
18         if (this == &string)
19         {
20             return *this;
21         }
22         if (str != NULL)             //释放内存
23         {
24             delete []str;
```

```
25            }
26            str = new char[strlen(string.str) + 1]; //申请内存
27            strcpy(str, string.str);              //复制字符串内容
28            return *this;
29        }
30
31        MyString & operator + (MyString &string)    //重载 +(改变被加
对象)
32        {
33            char *temp = str;
34            str = new char[strlen(temp) + strlen(string.str) + 1];
35            strcpy(str, temp);                     //复制第一个字符串
36            delete []temp;
37            strcat(str, string.str);              //连接第二个字符串
38            return *this;
39        }
40
41
42        /*MyString & operator + (MyString &string)    //重载 + (不改变
被加对象)
43        {
44            MyString *pString = new MyString("");    //堆内存中构造对
象
45            pString->str = new char[strlen(str) + strlen(string.str) + 1];
46            strcpy(pString->str, str);              //复制第一个字符串
47            strcat(pString->str, string.str);       //连接第二个字符串
48            return *pString;                         //返回堆中的对象
```

```
49      }*/
50
51      void print()                    //测试打印str成员
52      {
53          cout << str << endl;
54      }
55  private:
56      char *str;
57  };
58
59  /* //MyString类的友员，要求str成员是public访问权限
60  MyString & operator +(MyString &left, MyString &right)    //重载
+ (不改变被加对象)
61  {
62      MyString *pString = new MyString("");
63      pString->str = new char[strlen(left.str) + strlen(right.str) + 1];
64      strcpy(pString->str, left.str);
65      strcat(pString->str, right.str);
66
67      return *pString;
68  } */
69
70  int main(int argc, char* argv[])
71  {
72      MyString a("hello ");
73      MyString b("world");
74
```

```
75      MyString c("");
76      c = c + a;                      //先做加法，再赋值
77      c.print();
78      c = c + b;                      //先做加法，再赋值
79      c.print();
80
81      c = a + b;
82      a.print();
83      c.print();
84
85      return 0;
86  }
```

这里有3 个版本的 '+' 操作符重载函数，它们都是调用strcpy 复制第一个字符串，然后调用strcat连接第二个字符串。

第 1 个版本返回*this 对象，它改变了被加对象的内容。使用第一个 '+' 操作符重载函数版本的执行结果：

```
1  hello
2  hello world
3  hello world（对象a的str成员被改变了）
4  hello world
```

第 2 个版本和第 3 个版本都是返回堆中构造的对象，它们没有改变被加对象内容。它们的区别如下。

（1）第2个版本属于类的成员函数，而第3个版本是类的友员函数。

（2）第2个版本的参数为1个，而第3个版本的参数为2个，因为友员函数不含有this指针。

（3）由于类的友员函数不能使用私有成员，因此在这里使用第3个

版本时需要把str成员的访问权限改为public。

使用这两个 '+' 操作符重载函数版本的执行结果：

1　hello

2　hello world

3　hello（对象a的str成员没有被改变）

4　hello world

选择何种版本的 '+' 操作符重载函数要取决于实际情况。

面试题36 编程题——各类运算符重载函数的编写

考点：编写各类运算符重载函数

出现频率：★★★★

Implement a String class in C++ with basic functionality like comparison, concatenation, input and output. Please also provide some test cases and using scenarios (sample code of using this class).Please do not use MFC, STL and other libraries in your implementation.

（用C++实现一个String类，它具有比较、连接、输入、输出功能。并且请提供一些测试用例说明如何使用这个类。不能用MFC、STL以及其他库。）

【解析】

要实现本题要求的功能，需要重载下面的运算符。

（1）<、>、==和!=比较运算符。

（2）+= 连接运算符以及赋值运算符

（3）<<输出运算符以及>>输入运算符。

根据分析，可得到如下String类（String.h文件）的定义。

```
1    #ifndef STRING_H
2    #define STRING_H
3
4    #include <iostream>
5    using namespace std;
6
7    class String
```

```
8   {
9   public:
10       String();                             //默认构造函数
11       String(int n,char c);                 //普通构造函数
12       String(const char* source);           //普通构造函数
13       String(const String& s);              //复制构造函数
14       String& operator = (char* s);         //重载=,实现字符串赋值
15       String& operator = (const String& s);       //重载=,实现对象赋
值
16       ~String();                            //析构函数
17       char& operator[](int i);              //重载[],实现数组运算
18       const char& operator[](int i) const;       //重载[],实现数组运算
(对象为常量)
19       String& operator += (const String& s);       //重载+=,实现与字
符串相加
20       String& operator += (const char* s);       //重载+=,实现与对象
相加
21       friend ostream& operator << (ostream&out, String& s);       //重
载 <<,实现输出流
22       friend istream& operator >> (istream& in, String& s);       //重载
>>,实现输入流
23       friend bool operator < (const String& left,const String&
right);    //重载 <
24       friend bool operator > (const String& left, const String&
right);    //重载 >
25       friend bool operator == (const String& left, const String& right) ;
//重载 ==
```

```
26      friend bool operator != (const String& left, const String& right);   //重载 !=
27      char* getData();                        //获取data指针
28   private:
29      int size;                               //data表示的字符串长度
30      char* data;                             //指向字符串数据
31   };
32   #endif
```

为了实现与对象操作和与字符串操作，=和+=运算符的重载函数都有两个，它们的参数分别为String对象引用和字符指针。String.h文件的第21~26行声明运算符重载函数都是友员，并且这些函数所重载的运算符都是双目运算符，因此参数是两个。也可以把这些友员函数改为成员函数，此时参数是一个。还有一点需要注意：输入输出流操作符的重载最好是声明为友员函数。

String类声明的所有函数实现在String.cpp文件中，下面是String.cpp的清单。

```
1   #include "String.h"
2
3   String::String()                        //默认构造函数，构造空字符串
4   {
5         data = new char[1];               //空字符串只含有'\0'一个元素
6         *data = '\0';
7         size = 0;
8   }
9   String::String(int n,char c)            //普通构造函数
10   {                                      //含有n个相同字符的字符串
11      data = new char[n + 1];
```

```
12      size = n;
13      char *temp = data;                    //保存data
14      while(n--)                            //做n次赋值
15      {
16              *temp++ = c;
17      }
18          *temp = '\0';
19  }
20  String::String(const char *source)        //普通构造函数
21  {                                         //字符串内容与source相同
22      if (source == NULL)                   //source为NULL
23      {                                     //将data赋为空字符串
24              data = new char[1];
25              *data = '\0';
26          size = 0;
27      }
28      else
29      {                                     //source不为NULL
30          size = strlen(source);            //复制source字符串
31          data = new char[size + 1];
32          strcpy(data, source);
33      }
34  }
35  String::String(const String &s)           //复制构造函数
36  {                                         //字符串内容与对象s的相同
37      data = new char[s.size + 1];
38      strcpy(data, s.data);
```

```
39      size = s.size;
40  }
41  String& String::operator = (char *s)            // =重载
42  {                                       //目标为字符串
43      if (data != NULL)
44      {
45          delete []data;
46      }
47      size = strlen(s);
48      data = new char[size + 1];
49      strcpy(data, s);                        //复制目标字符串
50      return *this;
51  }
52  String& String::operator = (const String& s)    // =重载
53  {                                       //目标为String对象
55      {
54      if (this == &s)                         //如果对象s就是自己，直接返
回*this
56              return *this;
57      }
58      if (data != NULL)                       //释放data堆内存
59      {
60              delete []data;
61      }
62      size = strlen(s.data);
63      data = new char[size + 1];              //分配堆内存
64      strcpy(data, s.data);                   //复制对象s的字符串成员
```

```
65      return *this;
66  }
67  String::~String()
68  {
69      if (data != NULL)                   // data不为NULL，释放堆内
存
70      {
71          delete []data;
72          data = NULL;
73          size = 0;
74      }
75  }
76  char& String::operator [](int i)              //[]重载
77  {                                     //取数组下标为i的字符元素
78      return data[i];
79  }
80  const char& String::operator[] (int i) const
81  {
82      return data[i];
83  }
84  String& String::operator +=(const String& s)       // +=重载
85  {                                       //连接对象s的字符串成员
86      int len = size + s.size + 1;
87      char *temp = data;
88      data = new char[len];                    //申请足够的堆内存来存放
连接后的字符串
89      size = len - 1;
```

```
90      strcpy(data, temp);                        //复制原来的字符串
91      strcat(data, s.data);                  //连接目标对象内的字符串成
员
92      delete []temp;
93      return *this;
94    }
95    String& String::operator +=(const char *s)       //+=重载
96    {                                          //连接s字符串
97      if (s == NULL)
98      {
99         return *this;
100       }
101       int len = size + strlen(s) + 1;
102       char *temp = data;
103       data = new char[len];                     //申请足够的堆内存来存
放连接后的字符串
104       size = len - 1;
105       strcpy(data, temp);                       //复制原来的字符串
106       strcat(data, s);                        //连接目标字符串
107       delete []temp;
108       return *this;
109 }
110 String::length()                              //获取字符串长度
111 {
112       return size;
113 }
114 ostream& operator << (ostream &out, String &s)    //重载<<
```

```
115 {                                             //打印对象s内字符串成员的所有字符元素
116         for(int i = 0; i < s.length(); i++)
117         {
118             out << s[i] << " ";                  //输出字符串中每一个字符元素
119         }
120         return out;
121 }
122 istream& operator >> (istream& in, String& s)
123 {
124         char p[50];
125         in.getline(p, 50);                          //从输入流接收最多50个字符
126         s = p;                                      //调用赋值函数
127         return in;
128 }
129 bool operator < (const String& left, const String& right)    //重载 <
130 {
131         int i = 0;
132         while(left[i] == right[i] && left[i] != 0 && right[i] != 0)
133         {
134             i++;
135         }
136         return left[i]-right[i] < 0 ? true : false;
137 }
```

重载=、>、!=和重载<非常相似，这里就不列举了。

由于以上的代码清单中有详细的注释，因此这里就不再赘述了。另外还有两点需要说明： 友员函数不能访问String 类的私有成员，但由于重载了[]操作符，所以采取使用对象索引（left[i]和right[i]）的方式访问。

不能使用字符串复制（私有成员data 不能访问），而是调用赋值函数给对象s 赋字符串的内容（代码第126行）。

测试代码如下。

```
1    #include <iostream>
2    #include "String.h"
3    using namespace std;
4
5    int main(void)
6    {
7          String str(3, 'a');                    //普通构造函数测试
8          String str1(str);                      //复制构造函数测试
9          String str2("asdf");                   //普通构造函数
10       String str3;                           //默认构造函数测试
11
12       cout << "str: " << str << endl;
13       cout << "str1: " << str1 << endl;
14       cout << "str2: " << str2 << endl;
15       cout << "str3: " << str3 << endl;
16
17       str3 = str2;                           //赋值函数测试
18       cout << "str3: " << str3 << endl;
19       str3 = "12ab";                         //赋值函数测试
20       cout << "str3: " << str3 << endl;
```

```
21
22      cout << "str3[2] = " << str3[2] << endl;         //[]重载函数测试
23
24      str3 += "111";                                   //+=重载函数测试
25      cout << "str3: " << str3 << endl;
26      str3 += str1;                                    //+=重载函数测试
27      cout << "str3: " << str3 << endl;
28
29      cin >> str1;                                     // >>重载函数测试
30      cout << "str1: " << str1 << endl;
31
32      String t1 = "1234";
33      String t2 = "1234";
34      String t3 = "12345";
35      String t4 = "12335";
36
37      cout << "t1 == t2 ? " << (t1 == t2) << endl;      // ==重载函数
测试
38      cout << "t1 < t3 ? " << (t1 < t3) << endl;        // <重载函数测试
39      cout << "t1 > t4 ? " << (t1 > t4) << endl;        // >重载函数测试
40      cout << "t1 != t4 ? " << (t1 != t4) << endl;      // !=重载函数测
试
41
42      return 0;
43  }
```

测试程序执行结果：

```
1   str: a a a
```

```
2    str1: a a a
3    str2: a s d f
4    str3:
5    str3: a s d f
6    str3: 1 2 a b
7    str3[2] = a
8    str3: 1 2 a b 1 1 1
9    str3: 1 2 a b 1 1 1 a a a
10    123 456 abc def（终端输入）
11    str1: 1 2 3    4 5 6    a b c    d e f
12    t1 == t2 ? 1
13    t1 < t3 ? 1
14    t1 > t4 ? 1
15    t1 != t4 ? 1
```

面试题37 看代码写输出——new操作符重载的使用

考点：new操作符重载的使用

出现频率：★★

下面程序中主函数的new是类中new操作符重载。但是new后面只有一个参数0xa5，而类中函数的声明有两个参数。怎么会调用这个类的呢？

```
1   #include <malloc.h>
2   #include <memory.h>
3
4   class Blanks
5   {
6   public:
7           void *operator new( size_t stAllocateBlock, char chInit );
8   };
9
10   void *Blanks::operator new( size_t stAllocateBlock, char chInit )
11   {
12       void *pvTemp = malloc( stAllocateBlock );
13       if( pvTemp != 0 )
14           memset( pvTemp, chInit, stAllocateBlock );
15       return pvTemp;
16   }
17
```

```
18   int main()
19   {
20       Blanks *a5 = new( 0xa5 ) Blanks;
21
22       return a5 != 0;
23   }
```

【解析】

这里有以下几点需要说明。

重载 new 操作符第一个参数必须是 size_t 类型的，并且传入的值就是类的大小。本题中类的大小为1。如果类中含有一个int类型成员（int占4个字节），那么参数stAllocateBlock的值为4。

代码第20 行中的0xa5 表示第二个参数的大小，也就是chInit 为0xa5。

代码第14 行，用chInit 初始化分配的那块内存。

当执行代码第 20 行时，首先调用 Blanks 重载的 new 操作符函数，然后使用默认的构造函数初始化对象，最后用这个Blanks对象地址初始化a5。

第7章 C++继承和多态

继承和多态是 C++面向对象程序设计的关键。继承机制使得派生类能够获得基类的成员数据和方法，只需要在派生类中增加基类没有的成员。多态是建立在继承的基础上的，它使用了 C++编译器最核心的一个技术，即动态绑定技术。其核心思想是父类对象调用子类对象的方法。下面通过举例简单地说明继承的概念，如图7.1所示。

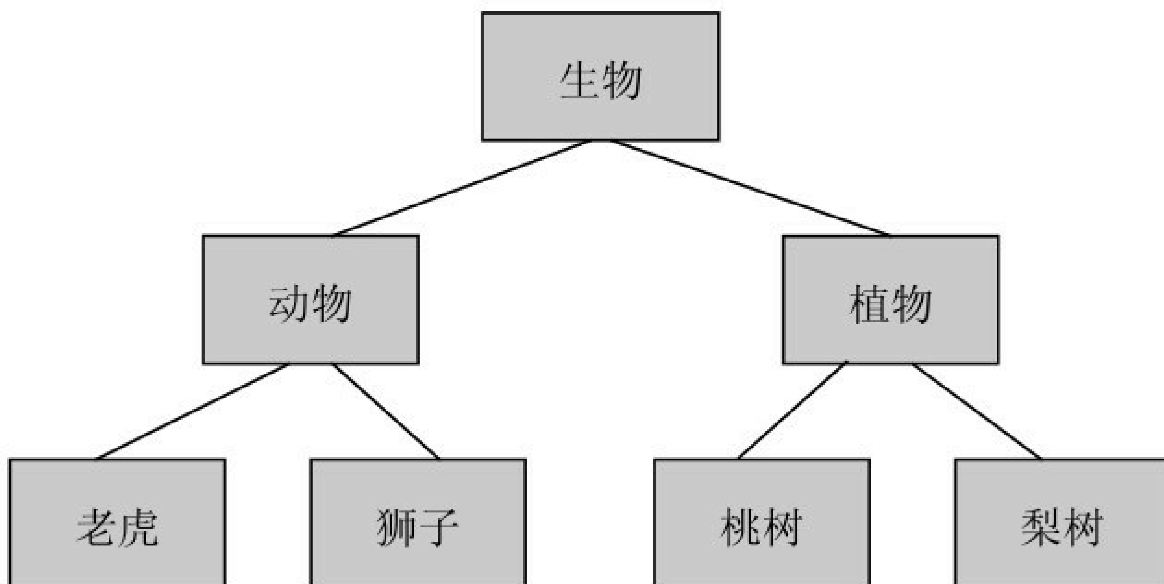


图7.1是一个抽象描述的特性继承表。

图7.1 继承的概念

生物是所有类的基类，所有生物都有寿命，所以可以把年龄作为生物类的属性。如果继续给生物分类，大家会想到有动物类和植物类等。当建立动物类和植物类的时候，无须再定义基类已经有的数据成员，而只需要描述动物类和植物类所特有的特性即可。比如动物类有奔跑、睡

觉等行为，往动物类添加相关的方法。

动物类和植物类的特性是由在生物类原有特性的基础上增加而来的，那么动物类和植物类就是生物类的派生类（也称作子类）。同样，老虎类和狮子类也是动物类的派生类，它们拥有动物类的一切特性。这种子类获得父类特性的概念就是继承。

面试题1 C++类继承的三种关系

考点：对C++类继承的三种关系的理解

出现频率：★★★★★

【解析】

C++中继承主要有三种关系：public、protected和private。

（1）public继承

public继承是一种接口继承，子类可以代替父类完成父类接口所声明的行为。此时子类可以自动转换成为父类的接口，完成接口转换。从语法角度上来说，public继承会保留父类中成员（包括函数和变量等）的可见性不变，也就是说，如果父类中的某个函数是 public的，那么在被子类继承后仍然是public的。

（2）protected继承

protected 继承是一种实现继承，子类不能代替父类完成父类接口所声明的行为，此时子类不能自动转换成为父类的接口。从语法角度上来说，protected继承会将父类中的public可见性的成员修改成为protected可见性，相当于在子类中引入了protected成员，这样在子类中同样还是可以调用父类的protected和public成员，子类的子类也就可以调用被protected继承的父类的protected和public成员。

（3）private继承

private 继承是一种实现继承，子类不能代替父类完成父类接口所声明的行为，此时子类不能自动转换成为父类的接口。从语法角度上来说，private 继承会将父类中的 public 和protected 可见性的成员修改成为private 可见性。这样一来，虽然子类中同样还是可以调用父类的protected和public成员，但是子类的子类就不可以再调用被private继承的

父类的成员了。

下面的程序代码说明了protected继承和private继承的区别。

```
1   #include <iostream>
2   using namespace std;
3
4   class Base
5   {
6   protected:
7       void printProtected() {cout << "print Protected" << endl;}
8   public:
9       void printPublic() {cout << "print Public" << endl;}
10  };
11
12  class Derived1 : protected Base        //protected继承
13  {
14  };
15
16  class Derived2 : private Base        //private继承
17  {
18  };
19
20  class A : public Derived1
21  {
22  public:
23      void print()
24      {
25          printProtected();
```

```
26          printPublic();
27      }
28  };
29
30  class B : public Derived2
31  {
32  public:
33      void print()
34      {
35          printProtected();           //编译错误，不能访问
36          printPublic();           //编译错误，不能访问
37      }
38  };
39
40  int main()
41  {
42      class A a;
43      class B b;
44      a.print();
45      b.print();
46      return 0;
47  }
```

Derived1 类通过 protected 继承 Base 类，因此它的派生类 A 可以访问 Base 基类的protected和public成员函数。

Derived2类通过private继承Base类，因此它的派生类B不可以访问Base基类的任何成员函数。

面试题2 C++继承关系

考点：对C++类继承的三种关系的理解

出现频率：★★★★

请考虑下面的标记为A～J的语句在编译时可能出现的情况。如果能够成功编译，请记为“RIGHT”，否则记为“ERROR”。

```
1    #include <iostream>
2    using namespace std;
3
4    class Parent
5    {
6    public:
7        Parent(int var = -1)
8        {
9            m_nPub = var;
10           m_nPtd = var;
11           m_nPrt = var;
12       }
13   public:
14       int m_nPub;
15   protected:
16       int m_nPtd;
17   private:
18       int m_nPrt;
19   };
```

```
20
21  class Child1 : public Parent
22  {
23  public:
24      int getPub() {return m_nPub;}
25      int getPtd() {return m_nPtd;}
26      int getPrt() {return m_nPrt;}    //A
27  };
28
29  class Child2 : protected Parent
30  {
31  public:
32      int getPub() {return m_nPub;}
33      int getPtd() {return m_nPtd;}
34      int getPrt() {return m_nPrt;}    //B
35  };
36
37  class Child3 : private Parent
38  {
39  public:
40      int getPub() {return m_nPub;}
41      int getPtd() {return m_nPtd;}
42      int getPrt() {return m_nPrt;}    //C
43  };
44
45
46  int main()
```

```
47  {
48      Child1 cd1;
49      Child2 cd2;
50      Child3 cd3;
51
52      int nVar = 0;
53
54      //public inherited
55      cd1.m_nPub = nVar;          //D
56      cd1.m_nPtd = nVar;          //E
57      nVar = cd1.getPtd();        //F
58      //protected inherited
59      cd2.m_nPub = nVar;          //G
60      nVar = cd2.getPtd();        //H
61      //private inherited
62      cd3.m_nPub = nVar;          //I
63      nVar = cd3.getPtd();        //J
64
65      return 0;
66  }
```

【解析】

A、B、C错误。m_nPrt是基类Parent的私有变量，不能被派生类访问。

D正确。Child1是public继承，可以访问并修改基类Parent的public成员变量。

E 错误。m_nPtd 是基类 Parent 的 protected 成员变量，通过公有继承后变成了派生类Child1的protected成员，因此只能在Child1类内部访

问，不能使用Child1对象访问。

F正确。可以通过Child1类的成员函数访问其protected变量。

G错误。Child2是protected继承，其基类Parent的public和protected成员变成了它的Protected成员，因此m_nPub只能在Child2类内部访问，不能使用Child2对象访问。

H正确。可以通过Child2类的成员函数访问其protected变量。

I错误。Child3是private继承，其基类Parent的public和protected成员变成了它的private成员，因此m_nPub只能在Child2类内部访问，不能使用Child2对象访问。

J正确。可以通过Child3类的成员函数访问其private变量。

【答案】

A、B、C、E、G、I为“ERROR”。

D、F、H、J为“RIGHT”。

面试题3 看代码找错——C++继承

考点：对C++类继承的三种关系的理解

出现频率：★★★★

```
1   #include <iostream>
2   using namespace std;
3
4   class base
5   {
6   private:
7       int i;
8   public:
9       base(int x) { i=x; }
10  };
11
12  class derived: public base
13  {
14  private:
15      int i;
16  public:
17      derived(int x, int y) { i=x;}
18      void printTotal()
19      {
20          int total = i + base::i;
21          cout << "total = " << total << endl;
```

```
22        }
23    };
24
25    int main()
26    {
27        derived d(1, 2);
28        d.printTotal();
29        return 0;
30    }
```

【答案】

这个程序有如下两个错误。

（1）在derived类进行构造时，它首先要调用其基类（base类）的构造方法，由于没有指明何种构造方法，因此默认调用base类不带参数的构造方法。然而，基类base中已经定义了带一个参数的构造函数，所以编译器就不会给它定义默认的构造函数了。因此代码第17行会出现“找不到构造方法”的编译错误。解决办法：可以在derived的构造函数中显示调用base的构造函数。

```
1     derived(int x, int y) : base(y) { i=x;}    //原代码第17行
```

（2）在derived类的printTotal()中，使用base::i的方式调用base类的私有成员i，这样会得到“不能访问私有成员”的编译错误。解决办法：把成员i的访问权限设为public。

面试题4 私有继承有什么作用

考点：对C++类私有继承的理解

出现频率：★★★★★

【解析】

先看下面的代码。

```
1    #include <iostream>
2    using namespace std;
3
4    class Person
5    {
6    public:
7        void eat() { cout << "Person eat" << endl;}
8    };
9
10   class Student : private Person    //私有继承
11   {
12   public:
13       void study() {cout << "Student Study" << endl;}
14   };
15
16   int main()
17   {
18       Person p;
19       Student s;
```

```
20
21        p.eat();
22        s.study();
23        s.eat();                //编译错误
24        p = s;                  //编译错误
25
26        return 0;
27    }
```

此程序的两个编译错误分别说明了私有继承的两个规则。

第一个规则正如大家现在所看到的，和公有继承相反，如果两个类之间的继承关系为私有，编译器一般不会将派生类对象（如Student）转换成基类对象（如Person）。这就是代码第24行失败的原因。

第二个规则是，从私有基类继承而来的成员都成为了派生类的私有成员—即使它们在基类中是保护或公有成员。这就是代码第23行失败的原因。

可以看出，私有继承时派生类与基类不是“is a”的关系，而是意味着“Is-Implement-In-Terms-Of”（以……实现）。如果使类D私有继承于类B，这样做是因为你想利用类B中已经存在的某些代码，而不是因为类B的对象和类D的对象之间有什么概念上的关系。因此，私有继承在软件“设计”过程中毫无意义，只是在软件“实现”时才有用。

面试题5 私有继承和组合有什么相同点和不同点

私有继承和组合有什么相同点和不同点？该如何选择？

考点：私有继承和组合的理解

出现频率：★★★

【解析】

使用组合表示“有一个（Has-A）”的关系。如果在组合中需要使用一个对象的某些方法，则完全可以利用私有继承代替。

私有继承下派生类会获得基类的一份备份，同时得到了访问基类的公共以及保护接口的权力和重写基类虚函数的能力。它意味着“以……实现（Is-Implement-In-Terms-Of）”，它是组合的一种语法上的变形（聚合或者“有一个”）。

例如“汽车有一个（Has-A）引擎”关系可以用单一组合表示，也可以用私有继承表示。例如下面的程序。

```
1    #include <iostream>
2    using namespace std;
3
4    class Engine
5    {
6    public:
7        Engine(int num) : numCylinders(num) {}    //Engine构造函数
8        void start()
9        {
10           cout << "Engine start, " << numCylinders << " Cylinders" <<
```

```
endl;
11      }
12  private:
13      int numCylinders;
14  };
15
16  class Car_pri : private Engine          //私有继承
17  {
18  public:
19      Car_pri() : Engine(8) {}          //调用基类的构造函数
20      void start()
21      {
22          Engine::start();              //调用基类的start()
23      }
24  };
25
26  class Car_comp
27  {
28  private:
29      Engine engine;                    //组合Engine类对象
30  public:
31      Car_comp() : engine(8) {}         //给engine成员初始化
32      void start()
33      {
34          engine.start();               //调用engine的start()
35      }
36  };
```

```
37
38  int main()
39  {
40      Car_pri car_pri;
41      Car_comp car_comp;
42
43      car_pri.start();
44      car_comp.start();
45
46      return 0;
47  }
```

由此看出，“有一个”关系既可以用私有继承表示，也可以用单一组合表示。

类Car_pri和类Car_comp有很多相似点：

（1）它们都只有一个Engine被确切地包含于Car中。

（2）它们在外部都不能进行指针转换，如将Car_pri *转换为Engine * 。

（3）它们都有一个start()方法，并且都在包含的Engine对象中调用start()方法。

也有下面一些区别：

（1）如果想让每个Car都包含若干Engine，那么只能用单一组合的形式。

（2）私有继承形式可能引入不必要的多重继承。

（3）私有继承形式允许Car 的成员将Car *转换成Engine *。

（4）私有继承形式允许访问基类的保护（protected）成员。

（5）私有继承形式允许Car重写Engine的虚函数。

应该在组合和私有继承之间如何选择呢？这里有一个原则：尽可能

使用组合，万不得已才用私有继承。请看下面的例子程序。

```
1   #include <iostream>
2   using namespace std;
3
4   struct Base                                  //抽象
5   {
6   public:
7     virtual void Func1() = 0;                  //纯虚函数
8     virtual void Func2() = 0;                  //纯虚函数
9     void print()
10      {
11        Func1(); //调用派生类的Fun1()
12        Func2(); //调用派生类的Fun1()
13      }
14   };
15
16   struct T : private Base
17   {
18   public:
19     virtual void Func1() {cout << "Func1" << endl;}    //覆盖基类的
Fun1
20     virtual void Func2() {cout << "Func2" << endl;}    //覆盖基类的
Fun2
21     void UseFunc()
22     {
23       Base::print();                          //调用基类的print()
24     }
```

```
25  };
26
27  int main()
28  {
29      T t;
30      t.UseFunc();
31      return 0;
32  }
```

程序输出如下。

```
1  Func1
2  Func2
```

上面的代码中Base类含有纯虚函数Fun1()以及Fun2()，因此它为抽象类，它通过虚函数调用了T中的重写版本。这种情况就不能使用组合了，因为组合的对象关系中不能使用一个抽象类，抽象类不能被实例化。

【答案】

相同点：都可以表示“有一个”关系。

不同点：私有继承中派生类能访问基类的protected成员，并且可以重写基类的虚函数，甚至当基类是抽象类的情况。组合不具有这些功能。

注意：选择它们的原则为尽可能使用组合，万不得已才用私有继承。

面试题6 什么是多态

考点：对C++多态的理解

出现频率：★★★★★

【解析】

多态（Polymorphism）、封装（Encapsulation）和继承（Inheritance）是面向对象思想的“三大特征”。可以说，不懂得什么是多态就不能说懂得面向对象。

多态性的定义：同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果。有两种类型的多态性：

（1）编译时的多态性。编译时的多态性是通过重载来实现的。对于非虚的成员来说，系统在编译时，根据传递的参数、返回的类型等信息决定实现何种操作。

（2）运行时的多态性。运行时的多态性就是指直到系统运行时，才根据实际情况决定实现何种操作。C++中，运行时的多态性通过虚成员实现。例如下面的程序代码。

```
1    #include <iostream>
2    using namespace std;
3
4    class Person
5    {
6    public:
7        virtual void print() {cout << "I'm a Person" << endl;}
8    };
9    class Chinese : public Person
```

```
10  {
11  public:
12      virtual void print() {cout << "I'm from China" << endl;}
13  };
14
15  class American : public Person
16  {
17  public:
18      virtual void print() {cout << "I'm from USA" << endl;}
19  };
20
21  void printPerson(Person &person)
22  {
23      person.print();       //运行时决定调用哪个类中的print()函数
24  }
25
26  int main()
27  {
28      Person p;
29      Chinese c;
30      American a;
31      printPerson(p);
32      printPerson(c);
33      printPerson(a);
34      return 0;
35  }
```

执行结果如下。

```
1    I'm a Person
2    I'm from China
3    I'm from USA
```

可以看到，在运行时通过基类Person的对象，可以来调用派生类Chinese和American中的实现方法。Chinese和American的方法都是通过覆盖基类中的虚函数方法来实现的。

面试题7 虚函数是怎么实现的

考点：C++虚函数实现的细节

出现频率：★★★★

【解析】

简单地说，虚函数是通过虚函数表实现的。那么，什么是虚函数表呢？

事实上，如果一个类中含有虚函数，则系统会为这个类分配一个指针成员指向一张虚函数表（vtbl），表中每一项指向一个虚函数的地址，实现上就是一个函数指针的数组。为了说明虚函数表，请看下面的程序用例。

```
1    class Parent
2    {
3    public:
4        virtual void foo1() {}
5        virtual void foo2() {}
6        void foo3();
7    };
8
9    class Child1
10   {
11   public:
12       void foo1() {}
13       void foo3();
14   };
```

```
15
16  class Child2
17  {
18  public:
19      void foo1() {}
20      void foo2() {}
21      void foo3();
22  };
```

下面列出了各个类的虚函数表（vtbl）的内容。

Parent类的vtbl：Parent::foo1()的地址、Parent::foo1()。

Child1类的vtbl：Child1::foo1()的地址、Parent::foo1()。

Child2类的vtbl：Child1::foo1()的地址、Child2::foo1()。

可以看出，虚函数表既有继承性，又有多态性。每个派生类的 vtbl 继承了它各个基类的vtbl，如果基类vtbl中包含某一项，则其派生类的vtbl中也将包含同样的一项，但是两项的值可能不同。如果派生类覆盖（override）了该项对应的虚函数，则派生类vtbl的该项指向重载后的虚函数，没有重载的话，则沿用基类的值。

在类对象的内存布局中，首先是该类的 vtbl 指针，然后才是对象数据。在通过对象指针调用一个虚函数时，编译器生成的代码将先获取对象类的vtbl指针，然后调用vtbl中对应的项。对于通过对象指针调用的情况，在编译期间无法确定指针指向的是基类对象还是派生类对象，或者是哪个派生类的对象。但是在运行期间执行到调用语句时，这一点已经确定，编译后的调用代码能够根据具体对象获取正确的 vtbl，调用正确的虚函数，从而实现多态性。

分析一下这里的思想所在，问题的实质是这样，对于发出虚函数调用的这个对象指针，在编译期间缺乏更多的信息，而在运行期间具备足够的信息，但那时已不再进行绑定了，怎么在二者之间做一个过渡呢？

把绑定所需的信息用一种通用的数据结构记录下来，该数据结构可以同对象指针相联系，在编译时只需要使用这个数据结构进行抽象的绑定，而在运行期间将会得到真正的绑定。这个数据结构就是vtbl。可以看到，实现用户所需的抽象和多态需要进行后绑定，而编译器又是通过抽象和多态实现后绑定的。

面试题8 构造函数调用虚函数

考点：C++虚拟机制的理解

出现频率：★★★★★

```
1   #include <stdio.h>
2
3   class A
4   {
5   public:
6       A() { doSth(); }              //构造函数调用虚函数
7       virtual void doSth() { printf("I am A"); }
8   };
9
10  class B : public A
11  {
12  public:
13      virtual void doSth() { printf("I am B"); }
14  } ;
15
16  int main()
17  {
18      B b;
19      return 0;
20  }
```

执行结果是什么？为什么？

【解析】

在构造函数中，虚拟机制不会发生作用，因为基类的构造函数在派生类构造函数之前执行，当基类构造函数运行时，派生类数据成员还没有被初始化。如果基类构造期间调用的虚函数向下匹配到派生类，派生类的函数理所当然会涉及本地数据成员，但是那些数据成员还没有被初始化，而调用涉及一个对象还没有被初始化的部分自然是危险的，所以C++会提示此路不通。因此，虚函数不会向下匹配到派生类，而是直接执行基类的函数。

【答案】

在构造函数中，虚拟机制不会发生作用，执行结果为：

```
1 I am A
```

面试题9 看代码写结果——虚函数的作用

考点：对C++类虚拟机制的理解

出现频率：★★★★★

```
1    #include <iostream>
2    using namespace std;
3    class A
4    {
5    public:
6        virtual void print(void)
7        {
8                cout << "A::print()" << endl;
9        }
10   };
11   class B:public A
12   {
13   public:
14       virtual void print(void)
15       {
16           cout << "B::print()" << endl;
17       }
18   };
19   class C:public A
20   {
21   public:
```

```
22      void print(void)
23      {
24          cout << "C::print()" << endl;
25      }
26  };
27  void print(A a)
28  {
29      a.print();
30  }
31  void main(void)
32  {
33      A a, *pa, *pb, *pc;
34      B b;
35      C c;
36
37      pa = &a;
38      pb = &b;
39      pc = &c;
40
41      a.print();
42      b.print();
43      c.print();
44
45      pa->print();
46      pb->print();
47      pc->print();
48
```

```
49      print(a);
50      print(b);
51      print(c);
52  }
```

【解析】

代码第41～43行，分别使用类A、类B和类C的各个对象来调用其print()成员函数，因此执行的是各个类的print()成员函数。

代码第45～47行，使用3个类A的指针来调用print()成员函数，而这3个指针分别指向类A、类B和类C的3个对象。由于print()函数是虚函数，因此这里有多态，执行的是各个类的print()成员函数。

代码第49～51行，全局的print()函数的参数使用传值的方式（注意与传引用的区别，如果是引用，则又是多态），在对象a、b、c分别传入时，在函数栈中会分别生成类A的临时对象，因此执行的都是类A的print()成员函数。

【答案】

执行结果如下。

```
1  A::print()
2  B::print()
3  C::print()
4  A::print()
5  B::print()
6  C::print()
7  A::print()
8  A::print()
9  A::print()
```

面试题10 看代码写结果——虚函数

考点：对C++类虚拟机制的理解

出现频率：★★★★★

```
1    #include <iostream>
2    #include <string>
3    using namespace std;
4
5    void println(const std::string& msg)
6    {
7        cout << msg << "\n";
8    }
9
10   class Base
11   {
12   public:
13       Base()
14       {
15           println("Base::Base()");
16           virt();
17       }
18       void f()
19       {
20           println("Base::f()");
21           virt();
```

```
22      }
23      virtual void virt()
24      {
25          println("Base::virt()");
26      }
27  };
28
29  class Derived : public Base
30  {
31  public:
32      Derived()
33      {
34          println("Derived::Derived()");
35          virt();
36      }
37      virtual void virt()
38      {
39          println("Derived::virt()");
40      }
41  };
42
43  int main(int argc,char* argv[])
44  {
45      Derived d;
46      Base *pB=&d;
47      pB->f();
48      return 0;
```

49　}

【解析】

代码第45行，构造Derived对象d。首先调用Base的构造函数，然后调用Derived的构造函数。在Base类的构造函数中，又调用了虚函数virt()，此时虚拟机制还没有开始作用（因为是在构造函数中），所以执行的是Base类的virt()函数。同样，在Derived类的构造函数中，执行的是Derived类的virt()函数。

代码第47行，通过Base类的指针pB访问Base类的公有成员函数f()。f()函数又调用了虚函数virt()，这里出现多态。由于指针pB是指向Derived类对象的，因此实际执行的是Derived类中的virt()成员。

【答案】

```
1　Base::Base()
2　Base::virt()
3　Derived::Derived()
4　Derived::virt()
5　Base::f()
6　Derived::virt()
```

面试题11 虚函数相关的选择题

考点：对C++类虚拟机制的理解

出现频率：★★★★

现有下面类和变量的定义。

```
1    #include <iostream>
2    #include <complex>
3    using namespace std;
4    class Base
5    {
6    public:
7        Base() { cout<<"Base-ctor"<<endl; }
8    ～Base() { cout<<"Base-dtor"<<endl; }
9        virtual void f(int) { cout<<"Base::f(int)"<<endl; }
10       virtual void f(double) {cout<<"Base::f(double)"<<endl; }
11       virtual void g(int i = 10) {cout<<"Base::g()"<<i<<endl; }
12   };
13
14   class Derived: public Base
15   {
16   public:
17       Derived() { cout<<"Derived-ctor"<<endl; }
18   ～Derived() { cout<<"Derived-dtor"<<endl; }
19       void f(complex<double> c){ cout<<"Derived::f(complex)"
<<endl;}
```

```
20        virtual void g(int i = 20) {cout<<"Derived::g()"<<i<<endl; }
21    };
22    Base b;
23    Derived d;
24    Base* pb = new Derived;
```

从4个选项中选择正确的一个。

（1）cout << sizeof(Base) << endl;

A．4

B．32

C．20

D．与平台相关

（2）cout << sizeof(Derived) << endl;

A．4

B．8

C．36

D．与平台相关

（3）pb->f(1.0);

A．Derived::f(complex)

B．Base::f(double)

C．Base::f(int)

D．Derived::f(double)

（4）pb->g();

A．Base::g()10

B．Base:g()20

C．Derived::g()10

D．Derived::g()20

【解析】

题（1），Base 类没有任何数据成员，并且含有虚函数，所以系统会为它分配一个指针指向虚函数表。指针的大小是4个字节。

题（2），Derived类没有任何数据成员，它是Base的派生类，因此它继承了Base的虚函数表。系统也会为它分配一个指针指向这张虚函数表。

题（3），Base 类中定义了两个 f()的重载函数，Derived 只有一个f()，其参数类型为complex，因此Derived并没有Base的f()进行覆盖。由于参数1.0默认是double类型的，因此调用的是Base:: f(double)。

题（4），Base和Derived都定义了含有相同参数列表的g()，因此这里发生多态了。pb指针指向的是Derived类的对象，因此调用的是Derived类的g()。这里要注意，由于参数的值是在编译期就已经决定的，而不是在运行期，因此参数i应该取Base类的默认值，即10。

【答案】

（1）A

（2）A

（3）B

（4）C

面试题12 为什么需要多重继承？它的优缺点是什么

考点：对C++多重继承的理解

出现频率：★★★★

【解析】

实际生活中，一些事物往往会拥有两个或两个以上事物的属性。为了解决这个问题， C++引入了多重继承的概念。C++允许为一个派生类指定多个基类，这样的继承结构被称作多重继承。举个例子：

人（Person）可以派生出作者（Author）和程序员（Programmer），然而程序员作者同时拥有作家和程序员的两个属性，即既能编程又能写作，如图7.2所示。

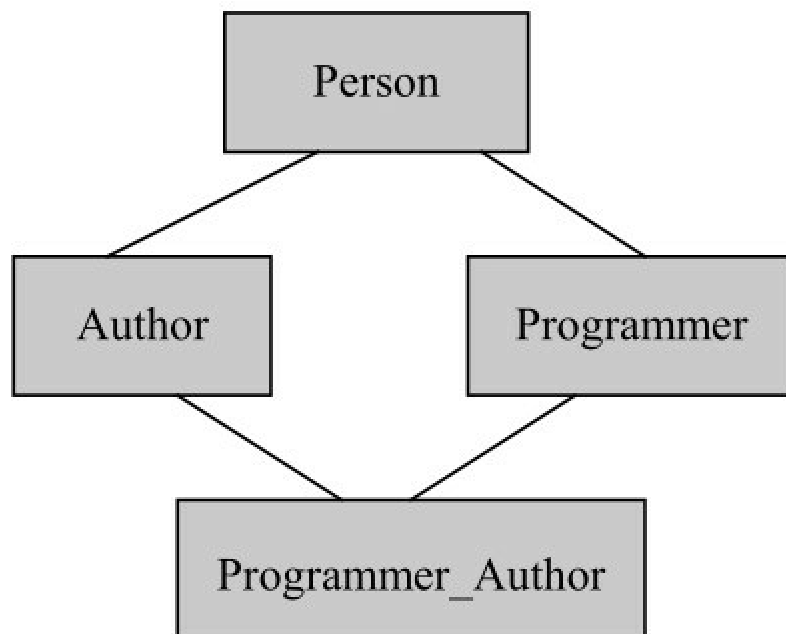


图7.2 各个类继承关系

使用多重继承的例子程序如下。

```
1   #include <iostream>
2   using namespace std;
3
4   class Person
5   {
6   public:
7       void sleep() {cout << "sleep" << endl;}
8       void eat() {cout << "eat" << endl;}
9   };
10
11   class Author : public Person        //Author继承自Person
12   {
13   public:
14       void writeBook() {cout << "write Book" << endl;}
15   };
16
17   class Programmer : public Person    //Programmer继承自Person
18   {
19   public:
20       void writeCode() {cout << "write Code" << endl;}
21   };
22
23   class Programmer_Author : public Programmer, public Author    //
多重继承
24   {
25   };
26
```

```
27  int main()
28  {
29      Programmer_Author pa;
30
31      pa.writeBook();    //调用基类Author的方法
32      pa.writeCode();    //调用基类Programmer的方法
33      pa.eat();        //编译错误，eat()定义不明确
34      pa.sleep();        //编译错误，sleep()定义不明确
35
36      return 0;
37  }
```

多重继承的优点很明显，就是对象可以调用多个基类中的接口，如代码第 31 行与代码第32行对象pa分别调用Author类的writeBook()函数和Programmer类的writeCode()函数。

多重继承的缺点是什么呢？如果派生类所继承的多个基类有相同的基类，而派生类对象需要调用这个祖先类的接口方法，就会容易出现二义性。代码第 33、34 行就是因为这个原因而出现编译错误的。因为通过多重继承的Programmer_Author类拥有Author类和 Programmer类的一份拷贝，而 Author类和 Programmer类都分别拥有Person类的一份拷贝，所以Programmer_Author类拥有Person类的两份拷贝，在调用Person类的接口时，编译器会不清楚需要调用哪一份拷贝，从而产生错误。对于这个问题，通常有两个解决方案：

（1）加上全局符确定调用哪一份拷贝。比如pa.Author::eat()调用属于Author的拷贝。

（2）使用虚拟继承，使得多重继承类Programmer_Author只拥有Person类的一份拷贝。比如在第11行和第17行的继承语句中加入virtual就可以了。

1　class Author : virtual public Person　　　//Author虚拟继承自Person

2　class Programmer : virtual public Person　　//Programmer虚拟继承自Person

【答案】

实际生活中，一些事物往往会拥有两个或两个以上事物的属性，为了解决这个问题， C++引入了多重继承的概念。

多重继承的优点是对象可以调用多个基类中的接口。

多重继承的缺点是容易出现继承向上的二义性。

面试题13 多重继承中的二义性

考点：对C++多重继承的理解

出现频率：★★★★★

下面程序中的多重继承有什么问题？

```
1    #include <iostream.h>
2    class cat
3    {
4    public:
5        void show()
6        {
7                cout<<"cat"<<endl;
8        }
9    };
10
11   class fish
12   {
13   public:
14       void show()
15       {
16           cout<<"fish"<<endl;
17       }
18   };
19
20   class catfish:public cat, public fish
```

```
21  {
22  };
23
24  int main()
25  {
26      catfish obj;
27      obj.show();
28
29      return 0;
30  }
```

【解析】

程序中catfish类多重继承cat类和fish类，因此继承了cat的show()方法和fish的show()方法。由于这两个方法同名，代码第 27 行直接用obj.show()时，无法区分应该执行哪个基类的show()方法，因此会出现编译错误。

【答案】

代码第27行出现编译错误，执行到obj.show()时无法区分应该执行哪个基类的show()方法，可以改成obj.car::show()访问car的show()成员。

面试题14 多重继承二义性的消除

考点：多重继承中二义性的消除

出现频率：★★★★

类A派生B和C，类D从B，C派生，如何将一个类A的指针指向一个类D的实例？

【解析】

这道题实际上考查的是如何消除多重继承引起的向上继承二义性问题。程序代码如下所示。

```
1    class A {};
2    class B : public A {};
3    class C : public A {};
4    class D : public B, public C {};
5
6    int main()
7    {
8        D d;
9        A *pd = &d; //编译错误
10       return 0;
11   }
```

由于B、C继承自A，B、C都拥有A的一份拷贝，类D多重继承自B、C，因此拥有A的两份拷贝。如果此时一个类A的指针指向一个类D的实例，会出现“模糊的转换”之类的编译错误。解决办法如下。

```
1    class A {};
2    class B : virtual public A {};      //B虚拟继承自A
```

```
3    class C : virtual public A {};        //C虚拟继承自A
4    class D : public B, public C {};
5
6    int main()
7    {
8        D d;
9        A *pd = &d;                  //成功转换
10       return 0;
11   }
```

将B、C都改为虚拟继承自A，则类D多重继承自B、C时，就不会重复拥有A的拷贝了，因此也就不会出现转换错误了。

【答案】

把B、C都改为虚拟继承自A，消除继承的二义性。

面试题15 多重继承和虚拟继承

考点：对多重继承和虚拟继承的理解

出现频率：★★★★

下面的程序输出结果是什么？

```
1    #include <iostream>
2    using namespace std;
3
4    class Parent
5    {
6    public:
7        Parent() : num(0) { cout << "Parent" << endl;}
8        Parent(int n) :num(n) {cout << "Parent(int)" << endl;}
9    private:
10       int num;
11   };
12   class Child1 : public Parent
13   {
14   public:
15       Child1() {cout << "Child1()" << endl;}
16       Child1(int num) : Parent(num) {cout << "Child1(int)" << endl;}
17   };
18   class Child2 : public Parent
19   {
20   public:
```

```
21      Child2() {cout << "Child2()" << endl;}
22      Child2(int num) : Parent(num) {cout << "Child2(int)" << endl;}
23  };
24  class Derived : public Child1, public Child2
25  {
26  public:
27      Derived() : Child1(0), Child2(1) {}
28      Derived(int num) : Child2(num), Child1(num+1) {}
29  };
30
31  int main()
32  {
33      Derived d(4);
34      return 0;
35  }
```

如果类Child1和Child2都改为virtual继承Parent，输出结果又是什么？

【解析】

首先讨论不存在virtual继承的情况。

多重继承类对象的构造顺序与其继承列表中基类的排列顺序一致，而不是与构造函数的初始化列表顺序一致。在这里，Derived继承的顺序是Child1、Child2（第24行），因此按照下面的步骤构造。

（1）构造Child1。由于Child1继承自Parent，因此先调用Parent的构造函数，再调用Child1的构造函数。

（2）调用Child2。过程与（1）类似，先调用Parent的构造函数，再调用Child2的构造函数。

（3）调用Derived类的构造函数。

因此输出结果为：

1　Parent(int)

2　Child1(int)

3　Parent(int)

4　Child2(int)

现在说明Child1和Child2都改为virtual继承Parent的情况。

当Child1和Child2为虚拟继承时，当系统碰到多重继承的时候就会自动先加入一个虚拟基类（Parent）的拷贝，即首先调用了虚拟基类（Parent）默认的构造函数，然后再调用派生类（Child1 和 Child2）的构造函数和自己（Derived）的构造函数。由于只生成一份拷贝，因此以后再也不会调用虚拟基类（Parent）的构造函数了，在 Child1 和 Child2 指定调用Parent的构造函数就无效了。

输出结果为：

1　Parent

2　Child1(int)

3　Child2(int)

现在来总结一下，多继承中的构造函数顺序如下：

（1）任何虚拟基类的构造函数按照它们被继承的顺序构造。

（2）任何非虚拟基类的构造函数按照它们被构造的顺序构造。

（3）任何成员对象的构造按照它们声明的顺序调用。

（4）类自身的构造函数。

【答案】

不存在virtual继承时的输出结果为：

1　Parent(int)

2　Child1(int)

3　Parent(int)

4　Child2(int)

存在 virtual继承时的输出结果为：

1 Parent
2 Child1(int)
3 Child2(int)

面试题16 为什么要引入抽象基类和纯虚函数

考点：对抽象基类和纯虚函数的理解

出现频率：★★★★★

【解析】

纯虚函数在基类中是没有定义的，必须在子类中加以实现，很像Java中的接口函数。如果基类含有一个或多个纯虚函数，那么它就属于抽象基类，不能被实例化。

为什么要引入抽象基类和纯虚函数呢？原因有以下两点：

（1）为了方便使用多态特性。

（2）在很多情况下，基类本身生成对象是不合情理的。例如，动物作为一个基类可以派生出老虎、狮子等子类，但动物本身生成对象明显不合常理。抽象基类不能够被实例化，它定义的纯虚函数相当于接口，能把派生类的共同行为提取出来。

以上面的动物、老虎、狮子类为例：

```
1    #include<iostream>
2    #include<memory.h>
3    #include<assert.h>
4    using namespace std;
5
6    class Animal
7    {
8    public:
9        virtual void sleep() = 0;    //纯虚函数，必须在派生类被定义
```

```
10      virtual void eat() = 0;   //纯虚函数，必须在派生类被定义
11   };
12
13   class Tiger : public Animal
14   {
15   public:
16      void sleep() {cout << "Tiger sleep" << endl;}
17      void eat() {cout << "Tiger eat" << endl;}
18   };
19
20   class Lion : public Animal
21   {
22   public:
23      void sleep() {cout << "Lion sleep" << endl;}
24      void eat() {cout << "Lion eat" << endl;}
25   };
26
27   void main()
28   {
29      Animal *p;            //Animal指针，不能使用Animal animal定
义对象
30      Tiger tiger;
31      Lion lion;
32
33      p = &tiger;           //指向Tiger对象
34      p->sleep();           //调用Tiger::sleep()
35      p->eat();             //调用Tiger::eat()
```

```
36      p = &lion;              //指向Lion对象
37      p->sleep();             //调用Lion::sleep()
38      p->eat();               //调用Lion::eat()
39  }
```

执行结果：

```
1   Tiger sleep
2   Tiger eat
3   Lion sleep
4   Lion eat
```

实际上，利用抽象类Animal把动物的共同行为抽出来了，那就是：不管是什么动物，都需要睡觉和吃食物。在上面的代码中，Animal 有两个纯虚函数分别对应这两个行为，因此 Animal 为抽象基类，不能被实例化。Animal 的两个纯虚函数 sleep()和 eat()在它的子类Tiger和Lion中都被定义了（如果子类中有一个基类的纯虚函数没有定义，那么子类也是抽象类）。虽然不能使用Animal animal 的方式生成Animal 对象，但可以使用Animal 的指针指向Animal的派生类Tiger和Lion，使用指针调用Animal类中的接口（纯虚函数）完成多态。

面试题17 虚函数与纯虚函数有什么区别？

考点：对虚函数和纯虚函数的理解

出现频率：★★★★

【解析】

虚函数和纯虚函数有以下方面的区别。

（1）类里如果声明了虚函数，这个函数是实现的，哪怕是空实现，它的作用就是为了能让这个函数在它的子类里面可以被覆盖，这样编译器就可以使用后期绑定来达到多态了。纯虚函数只是一个接口，是个函数的声明而已，它要留到子类里去实现。

（2）虚函数在子类里面也可以不重载；但纯虚函数必须在子类去实现，这就像Java的接口一样。通常把很多函数加上 virtual，是一个好的习惯，虽然牺牲了一些性能，但是增加了面向对象的多态性，因为很难预料到父类里面的这个函数不在子类里面不去修改它的实现。

（3）虚函数的类用于“实作继承”，也就是说继承接口的同时也继承了父类的实现。当然，大家也可以完成自己的实现。纯虚函数的类用于“介面继承”，即纯虚函数关注的是接口的统一性，实现由子类完成。

（4）带纯虚函数的类叫虚基类，这种基类不能直接生成对象，而只有被继承，并重写其虚函数后，才能使用。这样的类也叫抽象类。

面试题18 程序找错——抽象类不能实例化

考点：对抽象类不能实例化的理解

出现频率：★★★★★

```
1   #include<iostream>
2   using namespace std;
3
4   class Shape
5   {
6   public:
7       Shape() {}
8   ～Shape() {}
9       virtual void Draw() = 0;
10  }
11
12  void main()
13  {
14      Shape s1;
15  }
```

【答案】

Shape类的Draw()函数是一个纯虚函数，因此Shape类就是一个抽象类，它是不能实例化一个对象的。因此代码第14行出现编译错误。解决办法是把Draw函数修改成一般的虚函数或者把s1定义成Shape的指针。

面试题19 应用题——用面向对象的方法进行设计

考点：对面向对象编程的理解

出现频率：★★★

编写与一个图形相关的应用程序，需要处理大量图形（Shape）信息。图形有矩形（Rectangle）、正方形（Square）、圆形（Circle）等种类，应用需要计算这些图形的面积，并且可能需要在某个设备上进行显示（使用在标准输出上打印信息的方式作为示意）。

A．请用面向对象的方法对以上应用进行设计，编写可能需要的类。

B．请给出实现以上应用功能的示例性代码，从某处获取图形信息，并且进行计算和绘制。

C．Square是否继承自Rectangle？为什么？

【解析】

显然，不能说一个形状能有什么对象，而是说长方形或圆形等具体的图形类有对象。因此Shape为抽象类，其派生类有Rectangle和Circle等具体图形类。

那么定义形状（Shape）类有什么用处呢？显然，任何图形都有面积（Area），并且都能被显示（Draw），因此把这些共同的行为抽象出来作为Shape类的方法。由于Shape类为抽象类，因此这些方法在Shape类中就是纯虚函数。代码如下。

```
1    #include<iostream>
2    using namespace std;
3    #define PI 3.14159    //圆周率
```

```
4
5   //形状类
6   //////////////////////////////////////////////////////
7   class Shape
8   {
9   public:
10      Shape() {}
11   ～Shape() {}
12      virtual void Draw() = 0;        //纯虚函数
13      virtual double Area() = 0;        //纯虚函数
14   };
15   //////////////////////////////////////////////////////
16
17   //长方形类
18   //////////////////////////////////////////////////////
19   class Rectangle : public Shape
20   {
21   public:
22      Rectangle() : a(0), b(0) {}
23      Rectangle(int x, int y) : a(x), b(y) {}
24      virtual void Draw()
25      {
26          cout << "Rectangle, area: " << Area() << endl;
27      }
28      virtual double Area() {return a * b; }
29   private:
30      int a;
```

```
31      int b;
32  };
33  ////////////////////////////////////////////////////
34
35  //圆形类
36  ////////////////////////////////////////////////////
37  class Circle : public Shape
38  {
39  public:
40      Circle(double x) : r(x) {}
41      virtual void Draw()
42      {
43          cout << "Circle, area: " << Area() << endl;
44      }
45      virtual double Area() { return PI * r * r; }
46  private:
47      double r;
48  };
49  ////////////////////////////////////////////////////
50
51  //正方形类
52  ////////////////////////////////////////////////////
53  class Square : public Rectangle
54  {
55  public:
56      Square(int length) : a(length) {}
57      virtual void Draw()
```

```
58      {
59          cout << "Square, area: " << Area() << endl;
60      }
61      virtual double Area()
62      {
63          return a * a;
64      }
65  private:
66      int a;
67  };
68  ////////////////////////////////////////////////////
69
70  int main()
71  {
72      Rectangle rect(10, 20);
73      Square square(10);
74      Circle circle(8);
75
76      Shape *p;                    //抽象类指针
77      p = &rect;
78      cout << p->Area() << endl; //调用Rectangle::Area()
79      p->Draw();                 //调用Rectangle::Draw()
80
81      p = &square;
82      cout << p->Area() << endl; //调用Square::Area()
83      p->Draw();                 //调用Square::Draw()
84
```

```
85      p = &circle;
86      cout << p->Area() << endl; //调用Circle::Area()
87      p->Draw();              //调用Circle::Draw()
88
89      return 0;
90  }
```

在主函数中，使用了Shape类的指针去访问不同图形类的Draw()和Area()方法。这样， Shape类中的Draw()和Area()纯虚函数就被认为是接口，只要使用Shape类指针操作这些接口就可以了，而不用关心是子类中的具体实现。

实际上，正方形也可以直接继承自 Shape 类，但由于正方形可以看成是长和宽相等的长方形，可以认为是一种特殊的长方形，所以这里它继承自Rectangle类。

这样做的好处是操作方便，比如说在Rectangle中如果存在一个如下的虚函数：

```
1   virtual void foo() {cout << “Rectangle” << endl;}
```

注意，这个虚函数表示的是长方形的行为，而不是属于形状（Shape）的行为，并且如果这个foo()同时也属于正方形的行为，那么可以在Square类中对其进行覆盖。

```
1   virtual void foo() {cout << “Square” << endl;}
```

于是可以用Rectangle类指针操作Square类对象以达到多态。

当然，也会带来一些性能上的问题。大家知道，Square类继承Rectangle类，于是Square继承了Rectangle的虚表。如果Rectangle存在不同于Shape类的虚函数，则这张虚表所包括的项目就会增加。因此Square会有更多的虚表使用开销，导致程序执行效率上的下降。

面试题20 什么是COM

考点：对COM（组件对象模型）的理解

出现频率：★★★

【解析】

COM 即组件对象模型，是Component Object Model 取前3 个字母的缩写，这3 个字母在当今Windows的世界中随处可见。随时涌现出来的大把的新技术都以COM为基础。各种文档中也充斥着诸如COM对象、接口、服务器之类的术语。

简单地说，COM是一种跨应用和语言共享二进制代码的方法。与C++不同，它提倡源代码重用。源码级重用虽然好，但只能用于 C++。它还带来了名字冲突的可能性，更不用说不断拷贝重用代码而导致工程膨胀和臃肿。

Windows使用DLLs（动态链接库）在二进制级共享代码。这也是Windows程序运行的关键—重用kernel32.dll, user32.dll 等。但DLLs 是针对C 接口而写的，它们只能被C 或理解C调用规范的语言使用。由编程语言来负责实现共享代码，而不是由动态链接库本身。这样的话，动态链接库的使用受到限制。

COM通过定义二进制标准解决了这些问题。这是因为COM明确指出二进制模块（动态链接库和可执行文件）必须被编译成与指定的结构匹配。这个标准也确切地规定了在内存中如何组织COM对象。COM定义的二进制标准还必须独立于任何编程语言（如C++中的命名修饰）。一旦满足了这些条件，就可以轻松地从任何编程语言中存取这些模块。由编译器所负责产生的二进制代码与标准兼容。这样使后来的人就能更容易地使用这些二进制代码。

在内存中，COM对象的这种标准形式在C++虚函数中偶尔用到，所以这就是许多COM代码使用 C++的原因。但是记住，与编写模块所用的语言是无关的，因为结果二进制代码为所有语言可用。

【答案】

COM即组件对象模型，它定义了一种二进制标准，使得任何编程语言存取它所编写的模块。

面试题21 COM组件有什么特点

考点：对COM（组件对象模型）特点的理解

出现频率：★★★

【答案】

COM组件是遵循COM规范编写、以Win32动态链接库（DLL）或可执行文件（EXE）形式发布的可执行二进制代码，能够满足对组件架构的所有需求。遵循COM的规范标准，组件与应用、组件与组件之间可以互操作，极其方便地建立可伸缩的应用系统。COM是一种技术标准，其商业品牌则称为ActiveX。

组件在应用开发方面具有以下特点。

（1）组件是与开发工具语言无关的。开发人员可以根据特定情况选择特定语言工具实现组件的开发。编译之后的组件以二进制的形式发布，可跨Windows平台使用，而且源程序代码不会外泄，有效地保证了组件开发者的版权。

（2）通过接口有效保证了组件的复用性。一个组件具有若干个接口，每个接口代表组件的某个属性或方法。其他组件或应用程序可以设置或调用这些属性和方法来进行特定的逻辑处理。组件和应用程序的连接是通过其接口实现的。负责集成的开发人员无须了解组件功能是如何实现的，只需简单地创建组件对象并与其接口建立连接。在保证接口一致性的前提之下，可以调换组件、更新版本，也可以把组件安插在不同的应用系统中。

（3）组件运行效率高，便于使用和管理。因为组件是二进制代码，所以运行效率比ASP脚本高很多。核心的商务逻辑计算任务必须由组件来担当，ASP 脚本只起组装的角色。而且组件在网络上的位置可被

透明分配，组件和使用它的程序能在同一进程中、不同进程中或不同机器上运行。组件之间是相互独立的。组件对象通过一个内部引用计数器来管理它自己的生存期，这个计数器存放任何时候连接到该对象的客户数。当引用计数变为 0 时，对象可以把自己从内存中释放掉。这使程序员不必考虑与提供可共享资源有关的问题。

面试题22 如何理解COM对象和接口？

考点：对COM（组件对象模型）对象和接口的理解

出现频率：★★★

【答案】

一个对象实现一个接口，意思就是该对象使用代码实现了接口的每个方法并且为这些函数通向COM库提供了COM的二进制指针。然后COM使这些函数运行在请求了一个指向该接口的任何客户端。

COM在接口的定义和实现上有根本的差别。接口实际上是由一组定义了用法的相互联系的函数原型组成，只是它不能够被实现。这些函数原型就相当于 C++中含有纯虚拟函数的基类。

一个接口定义制定了接口的成员函数、调用方法、返回类型，它们的参数的类型和数量，以及这些函数要干什么。但是，这里并没有与接口实现相关的东西。

接口的实现就是程序员在一个接口定义上提供的执行相关动作的代码。客户调用完全决定于接口的定义。接口实现的一个实例，实际上就是一个指向一组方法的指针，即是指指向一个接口的函数表，该函数表引用了该接口所有方法的实现。每个接口是一个固定的一组方法的集合，在运行时通过globally unique interface identifier (IID)来定位。这里，IID是com支持的globally unique identifier (GUID)的特殊的实例。这样做就不会产生单一系统上相同名字、接口的多个版本的COM之间的冲突了。

一个COM接口与C++类是不一样的。一个COM接口不是一个对象，它只是简单地关联一组函数，是客户和程序之间通信的二进制标准。只要它提供了指向接口方法的指针，这个对象就可以用任何语言来

实现。COM接口是强类型的—每个接口有它自己的接口标识符。另外，不能用老版本的接口标识符定义新的版本，接口的IID定义的接口合同是明确、唯一的。

继承在COM里并不意味着代码的重用。因为接口没有实现关联，接口继承并不意味着代码继承。意思仅仅是，一个接口同一个合同关联，就像 C++的纯虚拟基类的创建和修改一样，可以添加方法或者更进一步的加强方法的使用。在COM里没有选择性继承。如果一个接口由另一个接口继承的话，它就包含了另一个接口定义的所有的方法。

管理实现一个COM对象的IUnknown::QueryInterface方法有3个主要规则：

（1）对象必须有一个标识符。

（2）一个对象实例的接口集合必须是静态的（static）。

（3）在对象中从任何一个其他的接口查询此接口都应该成功。

面试题23 简述COM、ActiveX和DCOM

考点：对COM、ActiveX以及DCOM的理解

出现频率：★★★

【答案】

COM（Component Object Mode）即组件对象模型，是组件之间相互接口的规范。其作用是使各种软件构件和应用软件能够用一种统一的标准方式进行交互。COM不是一种面向对象的语言，而是一种与源代码无关的二进制标准。

ActiveX 是 Microsoft 提出的一套基于 COM 的构件技术标准，实际上是对象嵌入与链接（OLE）的新版本。

基于分布式环境下的 COM 被称作 DCOM（Distribute COM，分布式组件对象模型），它实现了 COM 对象与远程计算机上的另一个对象之间直接进行交互。DCOM 规范定义了分散对象创建和对象间通信的机制，DCOM是ActiveX的基础，因为ActiveX主要是针对Internet应用开发（相比OLE）的技术，当然也可以用于普通的桌面应用程序。

面试题24 什么是DLL HELL

考点：对DLL HELL 的了解

出现频率：★★

【答案】

DLL HELL主要是指DLL（动态链接库）版本冲突的问题。一般情况下，DLL 新版本会覆盖旧版本，那么原来使用旧版本的DLL的应用程序就不能继续正常工作了。

扩展知识：虚函数表

大家知道，虚函数（Virtual Function）是通过一张虚函数表（Virtual Table）来实现的。在这个表中，主要是一个类的虚函数的地址表，这张表解决了继承、覆盖的问题，其内容真实反映实际的函数。这样，在有虚函数的类的实例中，这个表被分配在了这个实例的内存中，所以，当用父类的指针来操作一个子类的时候，这张虚函数表就显得尤为重要了。它就像一个地图一样，指明了实际所应该调用的函数。

C++的标准规格说明书中说到，编译器必须保证虚函数表的指针存在于对象实例中最前面的位置（这是为了保证正确取到虚函数的偏移量）。这意味着通过对象实例的地址得到这张虚函数表，然后就可以遍历其中的函数指针，并调用相应的函数。请看下面的程序例子。

```
1    #include <iostream>
2    using namespace std;
3
4    class Base
5    {
6    public:
```

```
7      virtual void fun1() {cout << "Base::fun1" << endl;}
8      virtual void fun2() {cout << "Base::fun2" << endl;}
9      virtual void fun3() {cout << "Base::fun3" << endl;}
10   private:
11       int num1;
12       int num2;
13   };
14
15   typedef void (*Fun)(void);
16
17   int main()
18   {
19       Base b;
20       Fun pFun;
21
22       pFun = (Fun)*((int*)*(int*)(&b)+0);   //取得Base::fun1()地址
23       pFun();                     //执行Base::fun1()
24       pFun = (Fun)*((int*)*(int*)(&b)+1);   //取得Base::fun1()地址
25       pFun();                     //执行Base::fun2()
26       pFun = (Fun)*((int*)*(int*)(&b)+2);   //取得Base::fun1()地址
27       pFun();                     //执行Base::fun3()
28
29       return 0;
30   }
```

上面程序的执行结果如下。

```
1    Base::fun1
2    Base::fun2
```

```
3    Base::fun3
```

可以看到，通过函数指针pFun的调用，分别执行了对象b的三个虚函数。通过这个示例发现，可以通过强行把&b 转成int *，取得虚函数表的地址，然后再次取址就可以得到第一个虚函数的地址了，也就是Base::fun1()。如果要调用Base::fun2()和Base::fun3()，只需要把&b先加上数组元素的偏移，后面的步骤类似就可以了。

程序中的Base对象b内存结构图如图7.3所示。

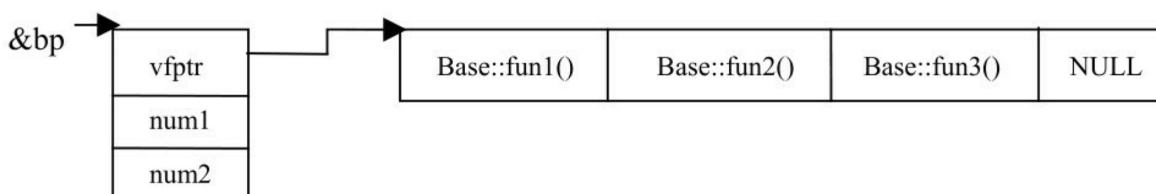


图7.3 Base 虚函数表图

一个类会有多少张虚函数表呢？

对于一个单继承的类，如果它有虚拟函数，则只有一张虚函数表。对于多重继承的类，它可能有多张虚函数表。

考虑下面代码中的各个类的定义。

```
1    #include <iostream>
2    using namespace std;
3
4    class Base1
5    {
6    public:
7        Base1(int num) : num_1(num) {}
8        virtual void foo1() {cout << "Base1::foo1 " << num_1 << endl;}
9        virtual void foo2() {cout << "Base1::foo2 " << num_1 << endl;}
10       virtual void foo3() {cout << "Base1::foo3 " << num_1 << endl;}
11   private:
```

```
12      int num_1;
13  };
14
15  class Base2
16  {
17  public:
18      Base2(int num) : num_2(num) {}
19      virtual void foo1() {cout << "Base2::foo1 " <<num_2 << endl;}
20      virtual void foo2() {cout << "Base2::foo2 " <<num_2 << endl;}
21      virtual void foo3() {cout << "Base2::foo3 " <<num_2 << endl;}
22  private:
23      int num_2;
24  };
25
26  class Base3
27  {
28  public:
29      Base3(int num) : num_3(num) {}
30      virtual void foo1() {cout << "Base3::foo1 " << num_3 << endl;}
31      virtual void foo2() {cout << "Base3::foo2 " << num_3 << endl;}
32      virtual void foo3() {cout << "Base3::foo3 " << num_3 << endl;}
33  private:
34      int num_3;
35  };
36
37  class Derived1 : public Base1
38  {
```

```
39  public:
40      Derived1(int num) : Base1(num) {}
41      virtual void faa1() {cout << "Derived1::faa1" << endl;}    //无覆
盖
42      virtual void faa2() {cout << "Derived1::faa2" << endl;}
43  };
44
45  class Derived2 : public Base1
46  {
47  public:
48      Derived2(int num) : Base1(num) {}
49      virtual void foo2() {cout << "Derived2::foo2" << endl;}    //只覆
盖了Base1::foo2
50      virtual void fbb2() {cout << "Derived2::fbb2" << endl;}
51      virtual void fbb3() {cout << "Derived2::fbb3" << endl;}
52  };
53
54  class Derived3 : public Base1, public Base2, public Base3    //多重
继承，无覆盖
55  {
56  public:
57      Derived3(int num_1, int num_2, int num_3) :
58              Base1(num_1), Base2(num_2), Base3(num_3) {}
59      virtual void fcc1() {cout << "Derived3::fcc1" << endl;}
60      virtual void fcc2() {cout << "Derived3::fcc2" << endl;}
61  };
62
```

63　class Derived4 : public Base1, public Base2, public Base3　　//多重继承，有覆盖

64　{

65　public:

66　　Derived4(int num_1, int num_2, int num_3) :

67　　　　Base1(num_1), Base2(num_2), Base3(num_3) {}

68　　virtual void foo1() {cout << "Derived4::foo1" << endl;}　//覆盖了Base1::foo1,

69　　　　　　　　　　　　　　　　　　　//Base2::foo1, Base3::foo1

70　　virtual void fdd() {cout << "Derived4::frr" << endl;}

71　};

这个例子说明了4种继承情况下的虚函数表。

1．一般继承（无虚函数覆盖）

Derived1类继承自Base1类，Derived1的虚函数表如图7.4所示。

Derived1类内没有任何覆盖基类Base1的函数，因此两个虚拟函数faa1()和faa2()被依次添加到了虚函数表的末尾。

2．一般继承（有虚函数覆盖）

Derived2类继承自Base1类，并对Base1类中的虚函数foo2()进行了覆盖。Derived2的虚函数表如图7.5所示。

Derived2 覆盖了基类 Base1 的 faa1()， 因此其虚函数表中Derived2::foo2()替换了Base1::foo2()一项，fbb2()和fbb3()被依次添加到了虚函数表的末尾。

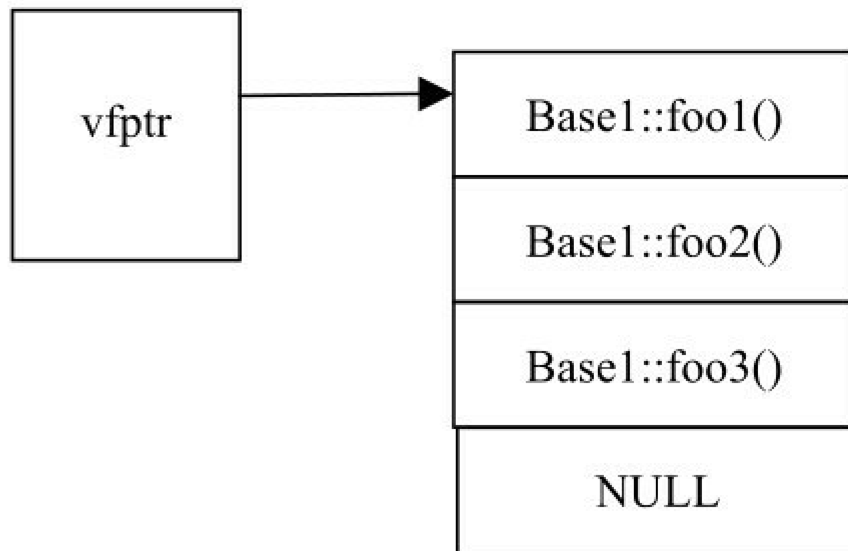


图7.4 Derived1虚函数表图

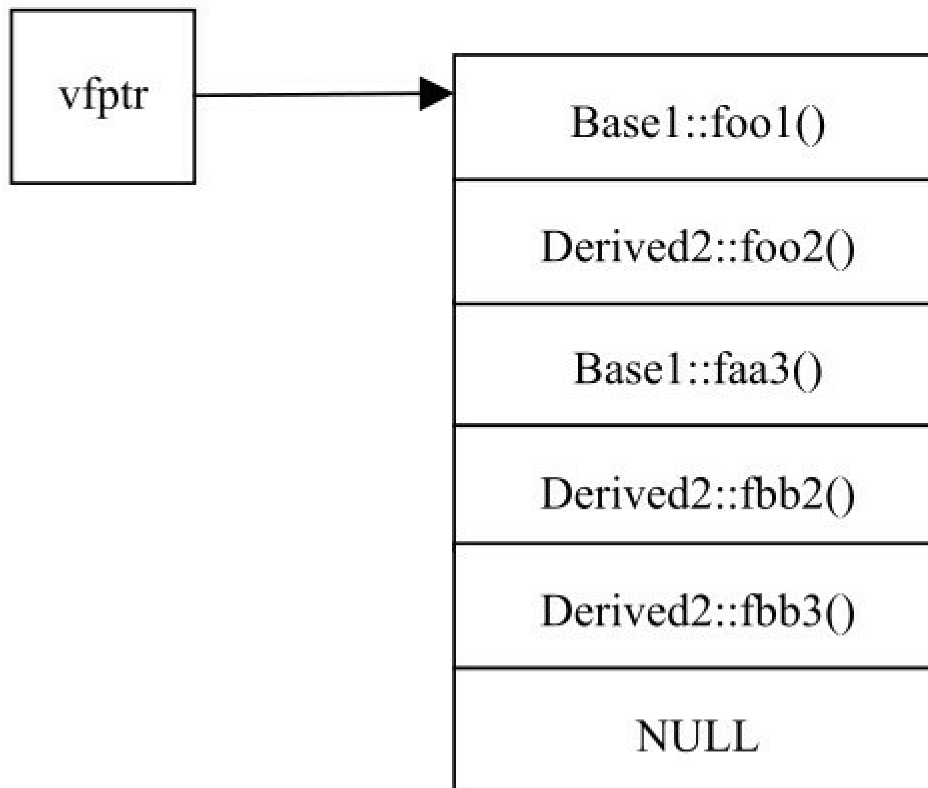


图7.5 Derived2虚函数表图

3．多重继承（无虚函数覆盖）

Derived3类继承自Base1类、Base2类、Base3类，Derived3的虚函数

表如图7.6所示。

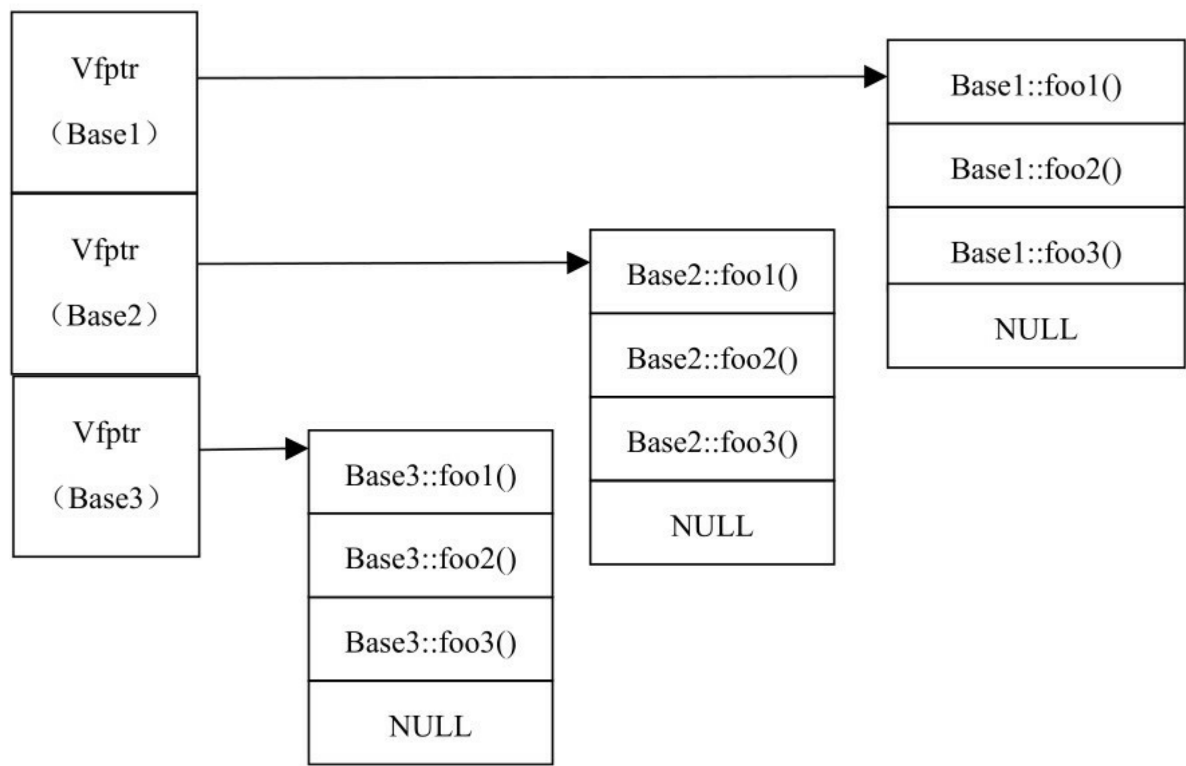


图7.6 Derived3虚函数表图

每个父类都有自己的虚表，Derived3 也就有了 3 个虚表，这里父类虚表的顺序与声明继承父类的顺序一致。这样做就是为了解决不同的父类类型的指针指向同一个子类实例，而能够调用到实际的函数。例如

```
1    Base2 *pBase2 = new Derived3();
2    pBase2->foo2();    //调用Base2::foo2()
```

把Base2类型的指针指向Derived3实例，那么调用将是对应Base2虚表里的那些函数。

4．多重继承（有虚函数覆盖）

Derived4类继承自Base1类、Base2类、Base3类，并对Base1类的foo1()、Base2类的foo1()、Base3类的foo1()都进行了覆盖。Derived4的虚函数表如图7.7所示。

可以看见，Base1::foo1()、Base2::foo1()和Base3::foo1()都被替换成

了Derived::foo1()。这样，我们就可以把任意一个静态类型的父类指向子类，并调用子类的f()了。如

```
1    Base1*pBase1 = new Derived4();
2    pBase1->foo1();          //调用从Base1继承的虚表中的Derived4::foo1()
```

下面是我们所讨论的四种继承情况下的测试代码。

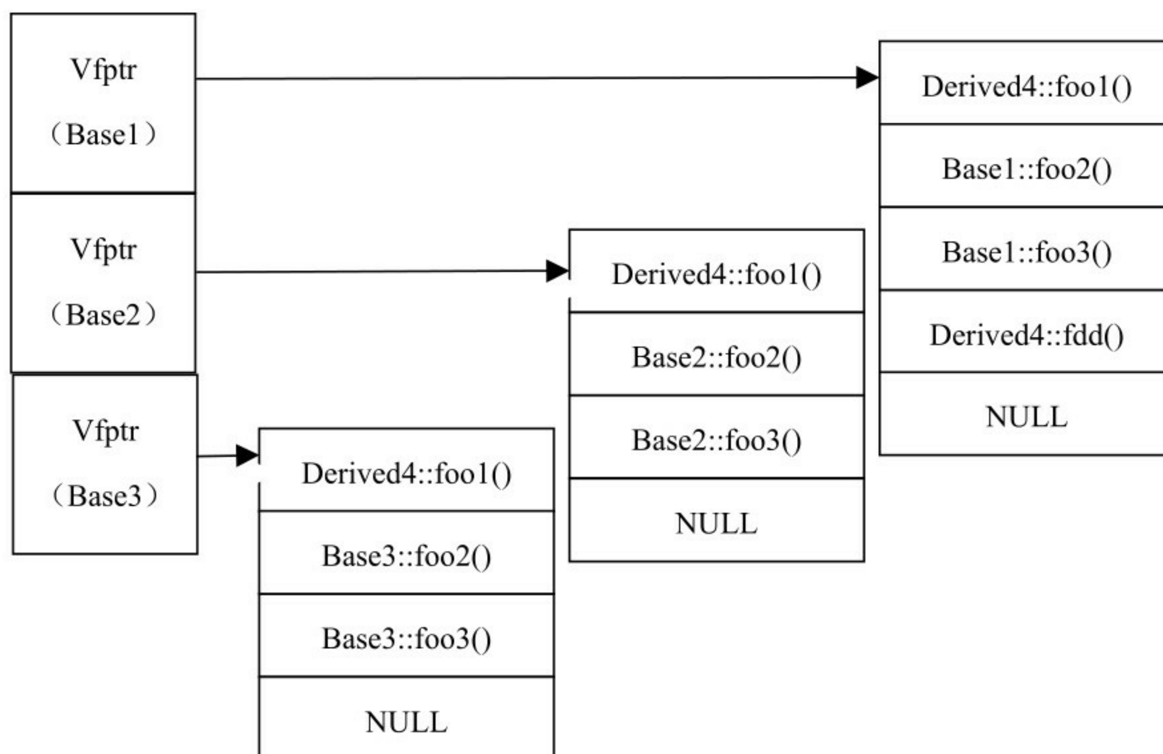


图7.7 Derived4虚函数表图

```
1    int main()
2    {
3        Base1 *pBase1 = NULL;
4        Base2 *pBase2 = NULL;
5        Base3 *pBase3 = NULL;
6
7        cout << "----- 一般继承自Base1，无覆盖    ----------" << endl;
```

```
8	Derived1 d1(1);		//Derived1一般继承自Base1，无覆盖
9	pBase1 = &d1;
10	pBase1->foo1();		//执行Base1::foo1();
11
12	cout << "----- 一般继承自Base1，覆盖foo2()	---------" <<
endl;
13	Derived2 d2(2);		//Derived2一般继承自Base1，覆盖了
Base1::foo2()
14	pBase1 = &d2;
15	pBase1->foo2();		//执行Derived2::foo2();
16
17	cout << "----- 多重继承，无覆盖	----------" << endl;
18	Derived3 d3(1, 2, 3);	//Derived3多重继承自
Base1,Base2,Base3,没有覆盖
19	pBase1 = &d3;
20	pBase2 = &d3;
21	pBase3 = &d3;
22	pBase1->foo1();		//执行Base1::foo1();
23	pBase2->foo1();		//执行Base2::foo1();
24	pBase3->foo1();		//执行Base3::foo1();
25
26	cout << "------ 多重继承，覆盖foo1()	---------" << endl;
27	Derived4 d4(1, 2, 3); //Derived4多重继承自Base1,Base2,Base3,
覆盖foo1()
28	pBase1 = &d4;
29	pBase2 = &d4;
30	pBase3 = &d4;
```

```
31        pBase1->foo1();       //执行Derived4::foo1();
32        pBase2->foo1();       //执行Derived4::foo1();
33        pBase3->foo1();       //执行Derived4::foo1();
34        return 0;
35    }
```

测试结果如下。

```
1   一般继承自Base1，无覆盖   ----------
2   Base1::foo1 1
3   一般继承自Base1，覆盖foo2()
4   Derived2::foo2
5   ----- 多重继承，无覆盖   ----------
6   Base1::foo1 1
7   Base2::foo1 2
8   Base3::foo3 3
9   ------ 多重继承，覆盖foo1()   ---------
10   Derived4::foo1
11   Derived4::foo1
12   Derived4::foo1
```

第8章 数据结构

数据结构主要研究数据的组织方式以及相应的操作方法。它除了描述数据本身之外，还描述数据之间的相互关系。它不仅是一般程序设计的基础，而且是设计编译程序、操作系统、数据库、人工智能及其他大型应用程序的基础。如今，数据结构在计算机科学中占有重要的地位。对于相当多的程序设计来说，认清数据的内在关系，可获得对问题的正确认识，看清问题的结构甚至解法。在一定意义上，程序所描述的就是在数据结构上实现的算法。算法的设计依赖于数据的逻辑结构，算法的实现依赖于数据的存储结构，所以数据结构选择得好坏，对程序质量的影响甚大。掌握基本的数据结构知识，是提高程序设计水平的必要条件。

单链表的结构是数据结构中最简单的，它的每一个节点只有一个指向后一个节点的指针，其模型如图8.1所示。

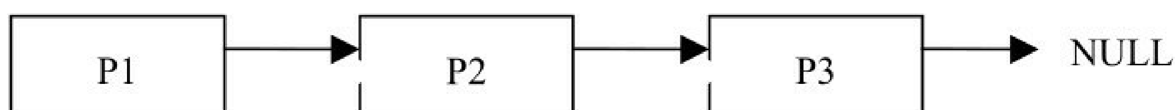


图8.1 单链表模型

循环链表与单链表一样，是一种链式的存储结构；不同的是，循环链表的最后一个节点的指针指向该循环链表的第一个节点或者表头节点，从而构成一个环形的链。其结构模型如图8.2所示。

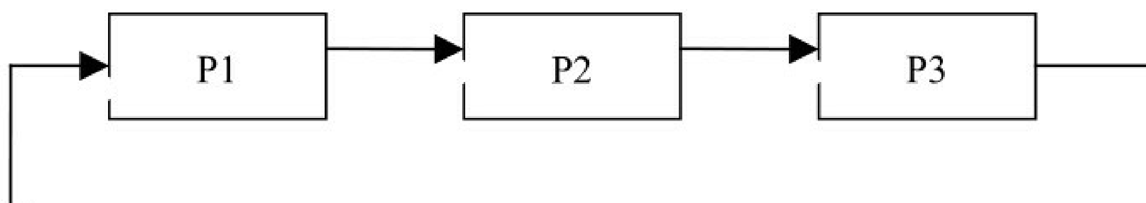


图8.2 循环链表模型

当对单链表进行操作时，有时你要对某个结点的直接前驱进行操作，又必须从表头开始查找。由于单链表每个结点只有一个存储直接后继结点地址的链域，因此运用单链表是无法办到的。那么能不能定义一个既有存储直接后继结点地址的链域，又有存储直接前驱结点地址的链域的这样一个双链域结点结构呢？有，这就是双向链表。

在双向链表中，结点除含有数据域外，还有两个指针，一个存储直接后继结点地址，另一个存储直接前驱结点地址。双向链表如图8.3所示。

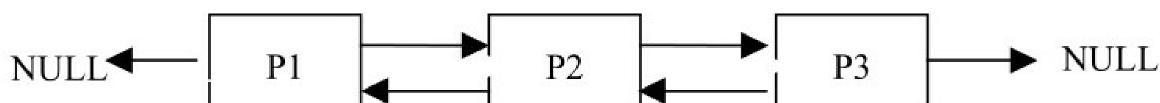


图8.3 双向链表模型

双向循环链表其实就是把双向链表的首尾相连，其模型图如图8.4所示。

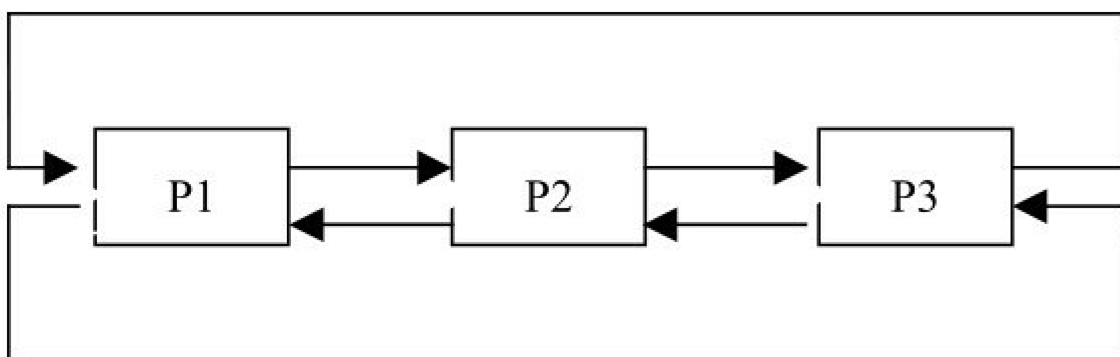


图8.4 双向循环链表模型

面试题1 编程实现一个单链表的建立

考点：单链表的操作

出现频率：★★★★

【解析】

链表节点的定义：

```
typedef struct node
{
   int data;           //节点内容
   node *next;        //下一个节点
}node;
```

单链表的创建：

```
1    //创建单链表
2    node *create()
3    {
4        int i = 0;                              //链表中数据的个数
5        node *head, *p, *q;
6        int x = 0;
7        head = (node *)malloc(sizeof(node));        //创建头节点
8
9        while(1)
10         {
11           printf("Please input the data: ");
12           scanf("%d", &x);
13           if (x == 0)                          //data为0时创建结束
```

```
14              break;
15          p = (node *)malloc(sizeof(node));
16          p->data = x;
17          if (++i == 1)
18          {                              //链表只有一个元素
19              head->next = p;            //连接到head的后面
20          }
21          else
22          {
23              q->next = p;               //连接到链表尾端
24          }
25          q = p;                         //q指向末节点
26      }
27      q->next = NULL;                    //链表的最后一个指针为
NULL
28      return head;
29  }
```

上面的代码中，使用 while 循环每次从终端读入一个整型数据，并调用 malloc 动态分配链表节点内存存储这个整型数据，然后插入到单链表的末尾。最后，当数据为 0 时表示插入数据结束，此时把末尾节点的next指针置为NULL。

面试题2 编程实现一个单链表的测长

考点：单链表的操作

出现频率：★★★★

【解析】

单链表的测长：

```
1 //返回单链表长度
2 int length(node *head)
3 {
4       int len = 0;
5       node *p;
6       p = head->next;
7       while(p != NULL)                    //遍历链表
8       {
9           len++;
10            p = p->next;
11        }
12        return len;
13 }
```

由于链表末尾节点的next指针被置为NULL，因此可以使用while循环遍历链表所有节点，当遇到NULL时结束循环。

面试题3 编程实现一个单链表的打印

考点：单链表的操作

出现频率：★★★★

【解析】

单链表的打印：

```
1   //打印单链表
2   void print(node *head)
3   {
4       node *p;
5       int index = 0;
6       if (head->next == NULL)          //链表为空
7       {
8           printf("Link is empty!\n");
9           return;
10      }
11      p = head->next;
12      while(p != NULL)              //遍历链表
13      {
14          printf("The %dth node is: %d\n", ++index, p->data);       //打印元素
15          p = p->next;
16      }
17  }
```

单链表的打印与单链表的测长方法类似，使用while循环遍历链表

所有节点并打印各个节点内容，当遇到NULL时结束循环。

面试题4 编程实现一个单链表节点的查找

考点：单链表的操作

出现频率：★★★★

【解析】

单链表节点的查找：

```
1   //查找单链表pos位置的节点,返回节点指针
2   //pos从0开始,0返回head节点
3   node *search_node(node *head, int pos)
4   {
5       node *p = head->next;
6       if (pos < 0)                      //pos位置不正确
7       {
8               printf("incorrect position to search node!\n");
9               return NULL;
10       }
11       if (pos == 0)                      //在head位置，返回head
12       {
13           return head;
14       }
15       if(p == NULL)
16       {
17           printf("Link is empty!\n");    //链表为空
18           return NULL;
19       }
```

```
20
21        while(--pos)
22        {
23            if ((p = p->next) == NULL)
24            {                              //超出链表返回
25                    printf("incorrect position to search node!\n");
26                    break;
27            }
28        }
29        return p;
30    }
```

面试题5 编程实现一个单链表节点的插入

考点：单链表的操作

出现频率：★★★★

【解析】

向单链表中某个位置（第pos个节点）之后插入节点，这里分为插入到链表首部、插入到链表中间，以及链表尾端3种情况。

```
1   //在单链表pos位置处插入节点，返回链表头指针
2   //pos从0开始计算,0表示插入到head节点后面
3   node *insert_node(node *head, int pos, int data)
4   {
5       node *item = NULL;
6       node *p;
7
8       item = (node *)malloc(sizeof(node));
9       item->data = data;
10      if (pos == 0)                   //插入链表头后面
11      {
12          head->next = item;          //head后面是item
13          return head;
14      }
15      p = search_node(head, pos);         //获得位置pos的节点指针
16      if (p != NULL)
17      {
18          item->next = p->next;       //item指向原pos节点的后一个
```

节点

```
19            p->next = item;            //把item插入到pos的后面
20        }
21        return head;
22    }
```

面试题6 编程实现一个单链表节点的删除

考点：单链表的操作

出现频率：★★★★

【解析】

单链表节点的删除：

```
1   //删除单链表的pos位置的节点，返回链表头指针
2   //pos从1开始计算，1表示删除head后的第一个节点
3   node *delete_node(node *head, int pos)
4   {
5       node *item = NULL;
6       node *p = head->next;
7       if (p == NULL)                    //链表为空
8       {
9           printf("link is empty!\n");
10          return NULL;
11      }
12      p = search_node(head, pos-1);       //获得位置pos的节点指针
13      if (p != NULL && p->next != NULL)
14      {
15          item = p->next;
16          p->next = item->next;
17          delete item;
18      }
19      return head;
```

```
20    }
```

下面是上面各个函数的测试程序。

```
1    int main()
2    {
3        node *head = create();                    //创建单链表
4        printf("Length: %d\n", length(head));    //测量单链表长度
5        head = insert_node(head, 2, 5);           //在第2个节点之后插入5
6        printf("insert integer 5 after 2th node: \n");
7        print(head);                              //打印单链表
8        head = delete_node(head, 2);              //删除第2个节点
9        printf("delete the 3th node: \n");
10       print(head);
11       return 0;
12   }
```

程序执行结果：

```
1    Please input the data: 1
2    Please input the data: 2
3    Please input the data: 3
4    Please input the data: 4
5    Please input the data: 0
6    Length: 4
7    insert integer 5 after 2th node:
8    The 1th node is: 1
9    The 1th node is: 2
10   The 1th node is: 5
11   The 1th node is: 3
12   The 1th node is: 4
```

```
13  delete the 3th node:
14  The 1th node is: 1
15  The 2th node is: 5
16  The 3th node is: 3
17  The 4th node is: 4
```

面试题7 实现一个单链表的逆置

考点：单链表的操作

出现频率：★★★★

【解析】

这是一个经常被问到的面试题，也是一个非常基础的问题。比如一个链表是这样的：1->2->3->4->5 通过逆置后成为5->4->3->2->1。

最容易想到的方法是遍历一遍链表，利用一个辅助指针，存储遍历过程中当前指针指向的下一个元素，然后将当前节点元素的指针反转后，利用已经存储的指针往后面继续遍历。

```
1   node *reverse(node *head)
2   {
3       node *p, *q, *r;
4
5       if (head->next == NULL)          //链表为空
6       {
7           return head;
8       }
9
10       p = head->next;
11       q = p->next;                //保存原第2个节点
12       p->next = NULL;                //原第1个节点为末节点
13
14       while(q != NULL)             //遍历，各个节点的next指针反转
15       {
```

```
16            r = q->next;
17            q->next = p;
18            p = q;
19            q = r;
20        }
21        head->next = p;                  //新的第1个节点为原末节点
22        return head;
23    }
```

面试题8 寻找单链表的中间元素

考点：单链表的操作

出现频率：★★★★

【解析】

这里使用一个只用一遍扫描的方法。描述如下：

假设mid指向当前已经扫描的子链表的中间元素，cur指向当前已扫描链表的末节点，那么继续扫描即移动cur到cur->next，这时只需判断一下应不应该移动mid到mid->next就行了。所以一遍扫描就能找到中间位置。代码如下。

```
1   node *search(node *head)
2   {
3       int i = 0;
4       int j = 0;
5       node *current = NULL;
6       node *middle = NULL;
7
8       current = middle = head->next;
9       while(current != NULL)
10        {
11            if( i / 2 > j)
12            {
13                    j++;
14                    middle = middle->next;
15            }
```

```
16          i++;
17          current = current->next;
18      }
19
20      return middle;
21  }
```

面试题9 单链表的正向排序

考点：单链表的操作

出现频率：★★★★

【解析】

结构体定义和代码如下。

```
1    typedef struct node
2    {
3        int data;
4        node *next;
5    }node;
6
7    node* InsertSort(void)
8    {
9        int data = 0;
10       struct node *head=NULL,*New,*Cur,*Pre;
11       while(1)
12       {
13           printf("please input the data\n");
14           scanf("%d", &data);
15           if (data == 0)                    //输入0结束
16           {
17               break;
18           }
19           New=(struct node*)malloc(sizeof(struct node));
```

```
20          New->data = data;                  //新分配一个node节点
21          New->next = NULL;
22          if(head == NULL)
23          {                              //第一次循环时对头节点赋值
24                  head=New;
25                  continue;
26          }
27          if(New->data <= head->data)
28          {//head之前插入节点
29              New->next = head;
30              head = New;
31              continue;
32          }
33          Cur = head;
34          while(New->data > Cur->data &&        //找到需要插入的位
置
35                  Cur->next!=NULL)
36          {
37              Pre = Cur;
38              Cur = Cur->next;
39          }
40          if(Cur->data >= New->data)           //位置在中间
41          {                              //把New节点插入到Pre和Cur之间
42                  Pre->next = New;
43                  New->next = Cur;
44          }
45          else                           //位置在末尾
```

```
46              Cur->next = New;            //把New节点插入到Cur之后
47      }
48      return head;
49  }
```

面试题10 判断链表是否存在环型链表问题

考点：单链表的操作

出现频率：★★★★

【解析】

这里有一个比较简单的解法。设置两个指针 p1、p2。每次循环 p1 向前走一步，p2 向前走两步。直到p2碰到NULL指针或者两个指针相等时结束循环。如果两个指针相等，则说明存在环。

程序代码如下。

```
1   //判断是否存在回环
2   //如果存在，start存放回环开始的节点
3   bool IsLoop(node* head, node **start)
4   {
5       node* p1=head, *p2 = head;
6
7       if (head == NULL || head->next ==NULL)
8       {                                   //head为NULL或
9           return false;                   //链表为空时返回false
10       }
11       do
12       {
13           p1 = p1->next;                  //p1走一步
14           p2 = p2->next->next;            //p2走两步
15       } while(p2 && p2->next && p1 != p2);
16
```

```
17        if(p1 == p2)
18        {
19            *start = p1;                    //p1为回环开始节点
20          return true;
21        }
22        else
23          return false;
24    }
25
26
27    int main()
28    {
29        bool bLoop = false;
30        node *head = create();                    //创建单链表
31        node *start = head->next->next->next;    //使第4个节点为回环开始位置
32        start->next = head->next;                //回环连接到第2个节点
33
34        node *loopStart = NULL;
35        bLoop = IsLoop(head, &loopStart);
36        printf("bLoop = %d\n", bLoop);
37        printf("bLoop == loopStart ? %d\n", (loopStart == start));
38        return 0;
39    }
```

main()函数中对IsLoop()函数做了测试，这里代码第31行到第32行手动把第2个节点接到了原来的第4个节点之后，于是节点4就成了回环开始的节点。因此，第36行和第37行的两条打印语句输出都是1。

面试题11 有序单链表的合并

考点：单链表的操作

出现频率：★★★★

已知两个链表 head1 和 head2 各自有序，请把它们合并成一个链表，依然有序。使用非递归方法以及递归方法。

【解析】

首先介绍非递归方法。因为两个链表 head1 和 head2 都是有序的，所以我们只需要把较短链表的各个元素有序地插入到较长的链表之中就可以了。

源代码如下。

```
1   node* insert_node(node *head, node *item)      //head != NULL
2   {
3       node *p = head;
4       node *q = NULL;                        //始终指向p之前的节点
5
6       while(p->data < item->data && p != NULL)
7       {
8           q = p;
9           p = p->next;
10      }
11      if (p == head)                        //插入到原头节点之前
12      {
13          item->next = p;
14          return item;
```

```
15       }
16       //插入到q与p之间
17       q->next = item;
18       item->next = p;
19       return head;
20   }
21
22   /* 两个有序链表进行合并 */
23   node* merge(node* head1, node* head2)
24   {
25       node* head;                        //合并后的头指针
26       node *p;
27       node *nextP;                       //指向p之后
28
29       if ( head1 == NULL )                  //有一个链表为空的情
况，直接返回另一个链表
30       {
31           return head2;
32       }
33       else if ( head2 == NULL )
34       {
35           return head1;
36       }
37
38       //两个链表都不为空
39       if(length(head1) >= length(head2))      //选取较短的链表
40       {                                  //这样进行的插入次数要少些
```

```
41            head = head1;
42            p = head2;
43        }
44        else
45        {
46            head = head2;
47            p = head1;
48        }
49
50        while(p != NULL)
51        {
52            nextP = p->next;                    //保存p的下一个节点
53            head = insert_node(head, p);        //把p插入到目标链表中
54            p = nextP;                          //指向将要插入的下一个节点
55        }
56
57        return head;
58    }
```

这里insert_node()函数是有序的插入节点，注意与前面例题中的函数有区别，这里它传入的参数是node*类型。然后在merge()函数中（代码第52～55行）循环把短链表中的所有节点插入到长链表中。

接下来介绍递归方法。比如有下面两个链表。

链表1：1->3->5

链表2：2->4->6

递归方法的步骤如下。

（1）比较链表1和链表2的第一个节点数据。由于1<2，因此把结果链表头节点指向链表1中的第一个节点，即数据1所在的节点。

（2）对剩余的链表1（3->5）和链表2再调用本过程，比较得到结果链表的第二个节点，即2与3比较得到2。此时合并后的链表节点为1->2。

接下来的过程类似（2），如此递归，直到两个链表的节点都被加到结果链表中。

```
1   node * MergeRecursive(node *head1, node *head2)
2   {
3       node *head = NULL;
4
5       if (head1 == NULL)
6       {
7               return head2;
8       }
9       if (head2 == NUL)
10        {
11            return head1;
12        }
13
14        if ( head1->data < head2->data )
15        {
16              head = head1 ;
17              head->next = MergeRecursive(head1->next,head2);
18        }
19        else
20        {
21              head = head2 ;
22              head->next = MergeRecursive(head1,head2->next);
```

```
23        }
24
25        return head ;
26    }
```

下面是测试程序。

```
1    int main()
2    {
3        node *head1 = create();           //创建单链表1
4        node *head2 = create();           //创建单链表2
5        //node *head = merge(head1, head2);
6        node *head = MergeRecursive(head1, head2);
7        print(head);
8
9        return 0;
10   }
```

这里使用merge()函数和MergeRecursive()函数测试，结果一致。

面试题12 约瑟夫问题的解答

考点：循环链表的操作

出现频率：★★★★

编号为1，2，…，N的N个人按顺时针方向围坐一圈，每人持有一个密码（正整数），一开始任选一个正整数作为报数上限值 M，从第一个人开始按顺时针方向自 1 开始按顺序报数，报到M时停止报数。报M的人出列，将他的密码作为新的M值，从他在顺时针方向上的下一个人开始重新从 1 报数，如此下去，直至所有人全部出列为止。试设计一个程序求出出列顺序。

【解析】

显然当有人退出圆圈后，报数的工作要从下一个人开始继续，而剩下的人仍然是围成一个圆圈的，因此可以使用循环单链表。由于退出圆圈的工作对应着表中结点的删除操作，对于这种删除操作频繁的情况，选用效率较高的链表结构。为了程序指针每一次都指向一个具体的代表一个人的结点而不需要判断，链表不带头结点。所以，对于所有人围成的圆圈所对应的数据结构采用一个不带头节点的循环链表来描述。设头指针为p，并根据具体情况移动。

为了记录退出的人的先后顺序，采用一个顺序表进行存储。程序结束后再输出依次退出的人的编号顺序。由于只记录各个节点的 data 值就可以，所以定义一个整型一维数组。如int quit[n];n 为一个根据实际问题定义的一个足够大的整数。

程序代码如下。

```
1    #include <iostream>
2    using namespace std;
```

```
3
4    /* 结构体和函数声明 */
5    typedef struct node
6    {
7        int data;
8        node *next;
9    } node;
10
11   node *node_create(int n);
12
13   //构造节点数量为 n的单向循环链表
14   node * node_create(int n)
15   {
16       node *pRet = NULL;
17
18       if (0 != n)
19       {
20           int n_idx    = 1;
21           node *p_node    = NULL;
22
23           /* 构造 n个 node */
24           p_node = new node[n];
25           if (NULL == p_node)        //申请内存失败，返回NULL
26           {
27               return NULL;
28           }
29           else
```

```
30          {
31              memset(p_node, 0, n * sizeof(node)); //初始化内存
32          }
33          pRet = p_node;
34          while (n_idx < n)           //构造循环链表
35          {                           //初始化链表的每个节点,从1到n
36                  p_node->data = n_idx;
37                  p_node->next = p_node + 1;
38                  p_node = p_node->next;
39                  n_idx++;
40          }
41          p_node->data = n;
42          p_node->next = pRet;
43      }
44
45      return pRet;
46  }
47
48  int main()
49  {
50      node *pList = NULL;
51      node *pIter = NULL;;
52      int n = 20;
53      int m = 6;
54
55      /* 构造单向循环链表 */
56      pList = node_create(n);
```

```
57
58      /* Josephus循环取数 */
59      pIter = pList;
60      m %= n;
61      while (pIter != pIter->next)
62      {
63          int i = 1;
64
65          /* 取到第 m-1个节点 */
66          for (; i < m - 1; i++)
67          {
68                  pIter = pIter->next;
69          }
70
71          /* 输出第 m个节点的值 */
72          printf("%d ", pIter->next->data);
73
74          /* 从链表中删除第 m个节点 */
75          pIter->next = pIter->next->next;
76          pIter = pIter->next;
77      }
78      printf("%d\n", pIter->data);
79
80      /* 释放申请的空间 */
81      delete []pList;
82      return 0;
83  }
```

面试题13 编程实现一个双向链表的建立

考点：双向链表的操作

出现频率：★★★★

【解析】

双向链表的定义如下。

```
1   typedef struct DbNode
2   {
3       int data;           //节点数据
4       DbNode *left;       //前驱节点指针
5       DbNode *right;   //后继节点指针
6   } DbNode;
```

（1）建立双向链表：为方便，这里定义了3个函数。

CreateNode()根据数据来创建一个节点，返回新创建的节点。

CreateList()函数根据一个节点数据创建链表的表头，返回表头节点。

AppendNode ()函数总在表尾插入新节点（其内部调用CreateNode()生成节点），返回表头节点。

```
1   //根据数据创建创建节点
2   DbNode *CreateNode(int data)
3   {
4       DbNode *pnode = (DbNode *)malloc(sizeof(DbNode));
5       pnode->data = data;
6       pnode->left = pnode->right = pnode;       //创建新节点时
7                                                  //让其前驱和后继指针都指向自身
```

```
8    return pnode;
9    }
10
11    //创建链表
12    DbNode *CreateList(int head)                //参数给出表头节点数据
13    {                                 //表头节点不作为存放有意义数据的节点
14        DbNode *pnode = (DbNode *)malloc(sizeof(DbNode));
15        pnode->data = head;
16        pnode->left = pnode->right = pnode;
17
18        return pnode;
19    }
20
21    //插入新节点,总是在表尾插入;返回表头节点
22    DbNode *AppendNode(DbNode *head, int data)    //参数1是链表的表头节点，
23    {                                 //参数2是要插入的节点,其数据为data
24        DbNode *node = CreateNode(data);          //创建数据为data的新节点
25        DbNode *p = head, *q;
26
27        while(p != NULL)
28        {
29            q = p;
```

```
30          p = p->right;
31      }
32      q->right = node;
33      node->left = q;
34
35      return head;
36  }
```

我们可以使用其中的CreateList()和AppendNode()来生成一个链表。下面是一个数据生成从0到9含有10个节点的循环链表。

```
1   DbNode *head = CreateList(0);              //生成表头，表头数据为0
2
3   for (int i = 1; i < 10; i++)
4   {
5     head = AppendNode(head, i);              //添加9个节点，数据为从1到9
6   }
```

面试题14 编程实现一个双向链表的测长

考点：双向链表的操作

出现频率：★★★★

【解析】

为了得到双向链表的长度，需要使用right指针进行遍历，直到得到NULL为止。

```
1   //获取链表的长度
2   int GetLength(DbNode *head)              //参数为链表的表头节点
3   {
4       int count = 1;
5       DbNode *pnode = NULL;
6
7       if (head == NULL)                    //head为NULL表示链表空
8       {
9           return 0;
10      }
11      pnode = head->right;
12      while (pnode != NULL)
13      {
14          pnode = pnode->right;            //使用right指针遍历
15          count++;
16      }
17
```

```
18        return count;
19    }
```

面试题15 编程实现一个双向链表的打印

考点：双向链表的操作

出现频率：★★★★

【解析】

与测长的方法一样，使用right指针进行遍历。

```
1    //打印整个链表
2    void PrintList(DbNode *head)                    //参数为链表的表头节点
3    {
4        DbNode *pnode = NULL;
5
6        if (head == NULL)                      //head为NULL表示链表空
7        {
8                return;
9        }
10       pnode= head;
11       while (pnode != NULL)
12       {
13           printf("%d ", pnode->data);
14           pnode = pnode->right;              //使用right指针遍历
15       }
16       printf("\n");
17   }
```

面试题16 编程实现一个双向链表节点的查找

考点：双向链表的操作

出现频率：★★★★

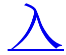
【解析】

使用right指针遍历，直至找到数据为data的节点。如果找到节点，返回节点，否则返回NULL。

```
1   //查找节点，成功则返回满足条件的节点指针，否则返回NULL
2   DbNode *FindNode(DbNode *head, int data)      //参数1是链表的表头节点
3   {                                             //参数2是要查找的节点,其数据为data
4       DbNode *pnode = head;
5
6       if (head == NULL)                //链表为空时返回NULL
7       {
8           return NULL;
9       }
10
11      /*找到数据或者到达链表末尾，退出while循环*/
12      while (pnode->right != NULL && pnode->data != data)
13      {
14          pnode = pnode->right;          //使用right指针遍历
15      }
```

```
16
17        //没有找到数据为data的节点，返回NULL
18        if (pnode->right == NULL)
19        {
20            return NULL;
21        }
22
23        return pnode;
24    }
```

面试题17 编程实现一个双向链表节点的插入



考点：双向链表的操作

出现频率：★★★★

【解析】

节点p后插入一个节点。

这里分为两种插入情况：一种是插入位置在中间，另一种是插入位置在末尾。两种情况有一点不同：插入位置在中间时需要把p的原后继节点的前驱指针指向新插入的节点。

```
1   //在node节点之后插入新节点
2   void InsertNode(DbNode *node, int data)
3   {
4       DbNode *newnode = CreateNode(data);
5       DbNode *p = NULL;
6
7       if (node == NULL)                    //node为NULL时返回NULL
8       {
9           return NULL;
10      }
11      if (node->right == NULL)             //node为最后一个节点
12      {
13          node->right = newnode;
14          newnode->left = node;
15      }
```

```
16      else
17      {                                   //node为中间节点
18          newnode->right = node->right;       //newnode向右连接
19          node->right->left = newnode;
20          node->right = newnode;          //newnode向左连接
21          newnode->left = node;
22      }
23  }
```

面试题18 编程实现一个双向链表节点的删除

考点：双向链表的操作

出现频率：★★★★

【解析】

这里有3种删除的情况：删除头节点、删除中间节点以及删除末节点。

下面的DeleteNode()删除数据为data的节点，并返回表头节点。如果不存在节点，则删除失败，返回NULL。如果删除后的链表为空，也返回NULL。

1 //删除满足指定条件的节点,返回表头节点,删除失败，返回NULL（失败的原因是不存在该节点）

2 DbNode *DeleteNode(DbNode *head, int data) //参数1是链表的表头节点

3 { //参数2是要插入的节点,其数据为data

4 DbNode *ptmp = NULL;

5

6 DbNode *pnode = FindNode(head, data); //查找节点

7 if (NULL == pnode) //节点不存在，返回NULL

8 {

9 return NULL;

10 }

11 else if (pnode->left == NULL) //node为第一个节点

```
12          {
13              head = pnode->right;
14              if (head != NULL)                   //链表不为空
15              {
16                      head->left = NULL;
17              }
18          }
19          else if (pnode->right == NULL)          //node为最后一个节点
20          {
21              pnode->left->right = NULL;
22          }
23          else                                    //node为中间的节点
24          {
25              pnode->left->right = pnode->right;
26              pnode->right->left = pnode->left;
27          }
28
29          free(pnode);                            //释放已被删除的节点空间
30          return head;
31      }
```

面试题19 实现有序双向循环链表的插入操作

考点：双向循环链表的操作

出现频率：★★★

【解析】

源代码如下。

```
1    #include <stdio.h>
2    #include <stdlib.h>
3
4    typedef struct DbNode
5    {
6        int data;                          //节点数据
7        DbNode *left;                      //前驱节点指针
8        DbNode *right;                     //后继节点指针
9    } DbNode;
10
11   //根据数据创建节点
12   DbNode *CreateNode(int data)
13   {
14       DbNode *pnode = (DbNode *)malloc(sizeof(DbNode));
15       pnode->data = data;
16       pnode->left = pnode;
17       pnode->right = pnode;              //创建新节点时
18                                          //让其前驱和后继指针都指向本身
```

```
19      return pnode;
20  }
21
22  //插入新节点,总是在表尾插入;返回表头节点
23  DbNode *AppendNode(DbNode *head, int data)    //参数1是链表
的表头节点
24  {                                   //参数2是要插入的节点,其数据为
data
25      DbNode *node = CreateNode(data);          //创建数据为data的
新节点
26      DbNode *p = NULL;
27      DbNode *q = NULL;
28
29      if (head == NULL)
30      {
31          return NULL;
32      }
33
34      q = p = head->right;
35      while(p != head)
36      {
37          q = p;
38          p = p->right;
39      }
40      q->right = node;                       //与原最后一个节点互连
41      node->left = q;
42      node->right = head;                    //与头节点互连，形成环状
```

```
43      head->left = node;
44
45      return head;
46  }
47
48  //打印整个链表
49  void PrintList(DbNode *head)                    //参数为链表的表头节
点
50  {
51      DbNode *pnode = NULL;
52
53      if (head == NULL)                        //head为NULL表示链表空
54      {
55          return;
56      }
57      printf("%d ", head->data);
58      pnode= head->right;
59      while (pnode != head)
60      {
61          printf("%d ", pnode->data);
62          pnode = pnode->right;                //使用right指针遍历
63      }
64      printf("\n");
65  }
66
67  //插入一个有序链表（从小到大排列），返回表头
68  DbNode *InsertNode(DbNode *head, int data)
```

```
69  {
70      DbNode *p = NULL, *q = NULL;
71      DbNode *node = NULL;
72
73      node = CreateNode(data);                    //新建数据节点
74      if (head == NULL)                          //空链表,返回新建节点
75      {
76          head =   node;
77          return node;
78      }
79
80      if (head->data > data)                    //data小于表头数据，插入到
表头之前
81      {                                        //把新建节点作为表头
82          head->left->right = node;            //末节点后继指针指向node
83          node->left = head->left;             //node的前驱指针指向末节点
84          node->right = head;                  //node的后继指针指向head
85          head->left = node;                   //head的前驱指针指向node
86          return node;
87      }
88
89      p = head->right;
90      while(p->data <= data && p != head)
91      {
92          p = p->right;
93      }
94
```

```
95        p = p->left;
96        q = p->right;
97
98        //把node插入p和q之间
99        p->right = node;
100        node->left = p;
101        node->right = q;
102        q->left = node;
103
104        return head;
105 }
106
107 int main()
108 {
109        DbNode *head = CreateNode(1);        //创建表头节点，其数
据为1
110        AppendNode(head, 3);              //添加节点3、6、8
111        AppendNode(head, 6);
112        AppendNode(head, 8);
113        PrintList(head);
114        head = InsertNode(head, 0);          //插入0，位置在开头
115        PrintList(head);
116        head = InsertNode(head, 4);          //插入4，位置在中间
117        PrintList(head);
118        head = InsertNode(head, 10);        //插入10，位置在末尾
119        PrintList(head);
120
```

```
121        return 0;
122 }
```

下面是各个函数的简单说明。

CreateNode()根据数据创建节点，包括头节点和一般节点。创建头节点时，left 和 right指针都指向本身来形成环状。

AppendNode()插入新节点，总是在末尾插入。返回表头节点。

PrintList()打印整个链表，由于属于循环链表，因此遍历回到head节点时结束打印。

InsertNode()数据插入一个升序的链表，返回表头节点。这里插入位置分为表头和非表头两种情况。如果是表头，返回的指针是新的节点，否则返回的是原来的表头节点。

main()函数建立了一个升序链表，并对它的不同位置进行了插入操作。

执行结果如下。

```
1    1 3 6 8
2    0 1 3 6 8
3    0 1 3 4 6 8
4    0 1 3 4 6 8 10
```

面试题20 删除两个双向循环链表的相同结点

考点：双向循环链表的操作

出现频率：★★★

有两个双向循环链表A，B，知道其头指针分别为pHeadA，pHeadB。请写一个函数，将两链表中data值相同的节点删除。

【解析】

可以这样来处理：

（1）把A中含有的与B中相同的数据节点抽出来，组成一个新的链表，例如

链表A：1 2 3 4 2 6 4

链表B：10 20 3 4 2 10

新建链表C: 2 3 4

（2）遍历链表C，删除A和B的所有节点。如果A含有数据相等的节点（A有两个4），且B也含有此数据节点（B也有4），则A中数据相等的节点全部被删除。

根据以上分析，我们实现了以下函数。

（1）GetLink()函数，其源代码如下。

1　//GetLink()通过headA链表和headB链表中相同的数据节点，得到一个新的链表

2　DbNode *GetLink(DbNode *headA, DbNode *headB)

3　{

4　　int i = 0;

5　　DbNode *newHead = NULL;　　　//返回新的链表

6 DbNode *pnodeA = headA; //遍历headA

7 DbNode *pnodeB = headB; //遍历headB

8 DbNode *pnode = NULL; //遍历newhead

9

10 do

11 {

12 DbNode *node = NULL; //用于查看新的链表中是否已经存在节点数据

13 pnodeB = headB;

14

15 if ((node = FindNode(newHead, pnodeA->data)) != NULL)

16 {

17 //若newHead中已含有此节点数据，则不考虑此节点的比较，进行下一次遍历

18 pnodeA = pnodeA->right;

19 continue;

20 }

21 do

22 {

23 if (pnodeA->data == pnodeB->data) //找到A与B中相同的数据节点

24 {

25 i++;

26 if (i == 1) //第一次生成头节点

27 {

28 newHead = CreateNode(pnodeA->data);

29 }

```
30                    else                //不是第一次，添加节点
31                    {
32                          AppendNode(newHead, pnodeA->data);
33                    }
34                    break;
35              }
36              pnodeB = pnodeB->right;          //对链表B进行遍历
37           } while(pnodeB != headB);
38           pnodeA = pnodeA->right;              //对链表A进行遍历
39        } while(pnodeA != headA);
40
41        return newHead;
42    }
```

GetLink()把A和B的两个链表头节点指针传入，返回一个链表头节点指针。这里有下面几点需要说明。

由于要遍历A和B，因此需要使用了嵌套的循环。外层是对链表A进行遍历的，内层是对链表B进行遍历的。

当找到A、B的相同数据后，由于新建表头节点情况（使用CreateNode函数）与插入节点情况（AppendNode函数）不同，因此代码第25～33行进行了区别对待。

出于效率的考虑，代码第15～20行首先判断A当前节点的数据是否已经在新建的链表中，如果已经存在，则不进行B的遍历，继续进行A的下一个节点的判断。

（2）DeleteNode函数，其源代码如下。

```
1    //删除链表中所有数据等于Value的节点
2    DbNode *DeleteNode(DbNode *pHeader, int Value)
3    {
```

```
4       DbNode *pNode = NULL;
5       DbNode *pNodeRight = NULL;
6       int bRet = 0;
7
8       if (pHeader == NULL)
9           return NULL;
10      while(pHeader->data == Value)        //头节点的数据为Value,
删除
11      {
12          pNode = pHeader->right;
13          if (pHeader == pHeader->right)      //链表只剩下一个元素
14          {
15              free(pHeader);             //删除节点，此时链表为空，
返回NULL
16              return NULL;
17          }
18          //链表还有其他节点，把原来头节点的左、右两个节点相
连
19          pHeader->left->right = pHeader->right;
20          pHeader->right->left = pHeader->left;
21          free(pHeader);               //释放头节点内存
22          pHeader = pNode;             //把原来头节点的下一个节点作
为新的头节点
23      }
24
25      pNode = pHeader->right;            //要删除的不是头节点
26      while(pNode != pHeader)            //遍历链表，直到回到头节
```

```
点
27        {
28            pNodeRight = pNode->right;        //保存下一个节点
29            if (pNode->data == Value)        //如果搜索到Value,删除此节
点
30            {
31                //把此节点的原来的左、右两个节点相连
32                pNode->left->right = pNodeRight;
33                pNodeRight->left = pNode->left;
34                free(pNode);
35            }
36            pNode = pNodeRight;                //指向下一个节点
37        }
38
39        return pHeader;
40    }
```

DeleteNode()传入的参数为链表头节点的指针以及需要删除的数据值，返回删除后的头节点指针。有下面几点需要说明。

当删除的节点为头节点时，此时需要考虑两种情况，即链表中只有一个头节点，以及删除后的头节点数据也等于Value值。代码第13～17行对第一种情况进行了判断，代码第10行的while循环是出于第二种情况的考虑。另外，头节点为下一个节点（pHeader发生了变化）。

当删除的节点不是头节点时，简单地进行循环，搜索数据为Value的节点并删除之，直到当前节点回到头节点结束。

（3）DeleteEqual ()函数，其源代码如下。

```
1    //DeleteEqual删除ppHeadA与ppHeadB两个链表所有含相同数据
的节点
```

2 //参数ppHeadA和ppHeadB分别为链表A和链表B头节点指针的地址

3 void DeleteEqual(DbNode **ppHeadA, DbNode **ppHeadB)

4 {

5 DbNode *pHeadA = *ppHeadA, *pHeadB = *ppHeadB;

6 DbNode *head = NULL;

7

8 if (ppHeadA == NULL || ppHeadB == NULL) //ppHeadA或ppHeadB不合法

9 {

10 return;

11 }

12 if (pHeadA == NULL || pHeadB == NULL) //链表A或B有一个为空

13 {

14 return;

15 }

16

17 head = GetLink(pHeadA, pHeadB); //获得A和B的相同数据节点所构成的链表

18 while(head != NULL)

19 {

20 pHeadA = DeleteNode(pHeadA, head->data); //删除A中所有head的数据节点

21 pHeadB = DeleteNode(pHeadB, head->data); //删除A中所有head的数据节点

22 head = DeleteNode(head, head->data); //删除head当前

节点

```
23      }
24
25      *ppHeadA = pHeadA;                          //返回ppHeadA
26      *ppHeadB = pHeadB;                          //返回ppHeadB
27  }
```

DeleteEqual()是最终的调用函数，它首先判断传入参数的有效性，然后调用前面的GetLink()得到链表A和链表B的相同数据所构成的新链表（代码第17行），最后调用前面的DeleteNode()循环删除各个链表中的节点（代码第18～23行）。

注意：由于DeleteEqual()函数有可能删除链表A和B的头节点，为了保存函数返回后A和B的头节点，参数必须使用指针的地址或者指针的引用（C++中可行）。这里使用的是指针的地址，所以在最后（代码第25、26行）保存链表头节点指针。

面试题21 编程实现队列的入队、出队、测长、打印

考点：队列的各项基本操作

出现频率：★★★★

【解析】

队列的实现可使用链表和数组，本题中我们使用单链表来实现队列。因此我们构造一个如图8.5所示结构的队列。

根据上面的结构图，我们可以使用下面的结构体定义队列和队列的节点。

```
1    typedef struct _Node
2    {
3        int data;
4        struct _Node *next; //指向链表下一个指针
5    } node;
6
7    typedef struct _Queue
8    {
9        node *front;                //队头
10       node *rear;                 //队尾
11   } MyQueue;
```

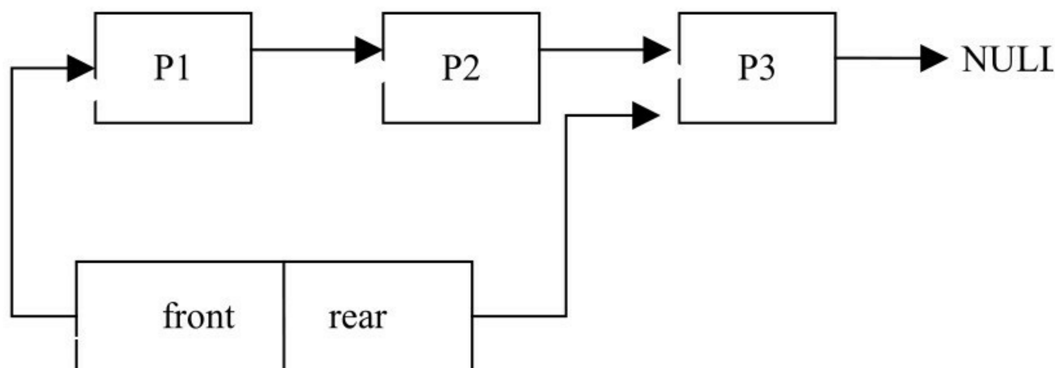


图8.5 链表实现队列

其中，node表示队列中的每个节点元素，MyQueue表示队列。

在进行本题要求的编程之前，我们首先要构造一个空的队列，代码如下。

```
1  //构造空的队列
2  MyQueue *CreateMyQueue()
3  {
4     MyQueue *q = (MyQueue *)malloc(sizeof(MyQueue));
5     q->front = NULL;       //把队首指针置空
6     q->rear = NULL;        //把队尾指针置空
7     return q;
8  }
```

CreateMyQueue函数中，把front和rear指针都置为NULL。

下面进行各个函数的编程。

1．入队

```
1  //入队，从队尾一端插入节点
2  MyQueue *enqueue(MyQueue *q, int data)
3  {
4     node *newP = NULL;
5     newP = (node *)malloc(sizeof(node));       //新建节点
```

```
6       newP->data = data;                    //复制节点数据
7       newP->next = NULL;
8       if (q->rear == NULL)
9       {
10          //若队列为空，则新节点既是队首又是队尾
11          q->front = q->rear = newP;
12        }
13        else
14        {
15          //若队列不为空，则新节点放到队尾，队尾指针指向新节点
16          q->rear->next = newP;
17          q->rear = newP;
18        }
19        return q;
20    }
```

由队列的特点可知，入队操作是在队尾一端进行插入的，enqueue函数的说明如下。

代码第5～7 行，根据数据data 新建一个数据节点。

代码第 8～12 行，如果队列为空，把新建的节点作为对首，同时也作为队尾。注意：此时不仅要操作rear指针，同时也需要对队列的front指针进行操作。

代码第13～18 行，如果队列不为空，把新建的节点连接到队尾，并将它作为新的队尾，此时只需要对front指针进行操作。

2．出队

```
1   //出队，从队头一端删除节点
2   MyQueue *dequeue(MyQueue *q)
```

```
3   {
4       node *pnode = NULL;
5       pnode = q->front;                  //指向队头
6       if (pnode == NULL)                  //队列为空
7       {
8           printf("Empty queue!\n");
9       }
10      else
11      {
12          q->front = q->front->next;      //新队头
13          if (q->front == NULL)           //当删除后队列为空时，对
rear置空
14          {
15              q->rear = NULL;
16          }
17          free(pnode);                    //删除原队头节点
18      }
19      return q;
20  }
```

出队与入队的操作不同，它是在队首一端进行删除的，dequeue函数的说明如下。

代码第5 行，指针pnode 指向队头节点，也就是即将要删除的指针节点。

代码第6～9 行，如果队列为空，打印“Empty queue”消息并返回。

代码第12～18 行，把队头节点往后移，然后删除原来的对头节点。如果删除后的队列为空，则把rear指针置空。

这里为简单起见，dequeue函数删除节点的同时释放了节点内存。

如果要得到被删除的节点，可以不释放内存并且返回此节点。

3．测长

```
1   //队列的测长
2   int GetLength(MyQueue *q)
3   {
4       int nlen = 0;
5       node *pnode = q->front;          //指向队头
6       if (pnode != NULL)               //队列不为空
7       {
8           nlen = 1;
9       }
10      while(pnode != q->rear)          //遍历队列
11      {
12          pnode = pnode->next;
13          nlen++;                      //循环一次，nlen递增1
14      }
15      return nlen;
16  }
```

GetLength函数的实现很简单，只需要遍历一次队列中的节点就可以获得。只有一点需要注意，在代码第10 行中，循环结束的条件是pnode != q->rear，而不应该与NULL 做比较（pnode != NULL）。这是因为队尾有可能指向的不是一个链表的末节点。

4．打印

```
1   //队列的打印
2   void PrintMyQueue(MyQueue *q)
3   {
4       node *pnode = q->front;
```

```
5       if (pnode == NULL)                  //队列为空
6       {
7             printf("Empty Queue!\n");
8             return;
9       }
10        printf("data: ");
11        while(pnode != q->rear)              //遍历队列
12        {
13           printf("%d ", pnode->data);    //打印节点数据
14           pnode = pnode->next;
15        }
16        printf("%d ", pnode->data);           //打印队尾节点数据
17    }
```

PrintMyQueue 函数的实现也很简单。与 GetLength 函数一样，都需要进行一次遍历队列中的节点，而且循环结束的条件也是pnode != q->rear，最后打印队尾节点数据（代码第16行）。

下面是对上面各个函数的简单测试。

```
1    int main()
2    {
3       int nlen = 0;
4       MyQueue *hp = CreateMyQueue();        //建立队列
5       enqueue(hp, 1); //入队 1 2 3 4
6       enqueue(hp, 2);
7       enqueue(hp, 3);
8       enqueue(hp, 4);
9       nlen = GetLength(hp);               //获得队列长度
10        printf("nlen = %d\n", nlen);
```

```
11        PrintMyQueue(hp);                    //打印队列数据
12        dequeue(hp);                         //出队两次
13        dequeue(hp);
14        nlen = GetLength(hp);                //再次获得队列长度
15        printf("\nnlen = %d\n", nlen);
16        PrintMyQueue(hp);                    //再次打印队列数据
17        return 0;
18    }
```

执行结果如下。

```
1    nlen = 4
2    data: 1 2 3 4
3    nlen = 2
4    data: 3 4
```

面试题22 队列和栈有什么区别

考点：队列和栈的区别

出现频率：★★★★★

【解析】

队列与栈是两种不同的数据结构。它们有以下区别。

（1）操作的名称不同。队列的插入称为入队，队列的删除称为出队。栈的插入称为进栈，栈的删除称为出栈。

（2）可操作的方向不同。队列是在队尾入队，队头出队，即两边都可操作。而栈的进栈和出栈都是在栈顶进行的，无法对栈底直接进行操作。

（3）操作的方法不同。队列是先进先出（FIFO），即队列的修改是依先进先出的原则进行的。新来的成员总是加入队尾（不能中间插入），每次离开的成员总是队列头上的（不允许中途离队）。而栈为后进先出（LIFO），即每次删除（出栈）的总是当前栈中“最新的”元素，即最后插入（进栈）的元素，而最先插入的被放在栈的底部，要到最后才能删除。

面试题23 简答题——队列和栈的使用

考点：队列和栈的使用

出现频率：★★★★★

一个顺序为1，2，3，4，5，6的栈，依次进入一个队列，然后进栈，顺序是什么？

【解析】

首先一个顺序为1，2，3，4，5，6的栈，其意思是进栈的顺序是1，2，3，4，5，6。按照栈的结构，1由于最先进栈，所以被放入栈底；6最后进栈，因此6位于栈顶。

然后进入一个队列。因为只能在栈顶进行出栈操作，也就是说，6最先出栈，1最后出栈。因此，队列的入队顺序（也就是栈的出栈顺序）为6，5，4，3，2，1。

最后再进栈。队列是个FIFO（先进先出）的结构，因此出队顺序与入队的顺序相同，即6，5，4，3，2，1。也就是6最先进栈，1最后进栈。因此，此时6位于栈底，1位于栈顶。

【答案】

最后的入栈顺序为6，5，4，3，2，1。

面试题24 选择题——队列和栈的区别

考点：队列和栈的区别

出现频率：★★★★

Which one is correct according to Stack and Queue?

A. Stack is FILO, while Queue is FIFO

B. Stack is FIFO, while Queue is FILO

C. Both Stack and Queue are FILO

D. Both Stack and Queue are FIFO

【解析】

本题考查的是栈与队列的区别，栈是先进后出（FILO），而队列是先进先出（FIFO）。

【答案】

A

面试题25 使用队列实现栈

考点：队列的使用以及栈的实现

出现频率：★★★★

编程实现下面的stack，并根据这个stack完成queue的操作。

```
1    class MyStack
2    {
3        void push(data);
4        void pop(&data);
5        bool isEmpty();
6    };
```

【解析】

首先说明这个stack的实现。显然，我们需要实现栈的3种基本操作，即进栈、出栈以及判空。为方便起见，这里使用单链表结构实现栈并且使用类的形式来定义栈及其节点。首先是节点类和栈类的具体定义，源代码如下。

```
1    class MyData
2    {
3    public:
4        MyData() : data(0), next(NULL) {}                    //默认构造函数
5        MyData(int value) : data(value), next(NULL) {}    //带参数的构造函数
6        int data;                                     //数据域
7        MyData *next;                                   //下一个节点
8    };
```

```
9
10  class MyStack
11  {
12  public:
13      MyStack() : top(NULL) {}                        //默认构造函数
14      void push(MyData data);                         //进栈
15      void pop(MyData *pData);                        //出栈
16      bool IsEmpty();                                 //是否为空栈
17      MyData *top;                                    //栈顶
18  };
```

这里MyData定义了单链表的节点，其中data表示节点的数据域，next指向下一个节点。MyStack表示栈的定义，其中private成员top表示栈顶，由于不能直接操作栈底，因此这里没有定义栈底的指针。在MyStack的默认构造函数中，把栈顶指针top置空，表示此时栈为空栈。

接下来是进栈、出栈以及判空的代码实现。

```
1  //进栈
2  void MyStack::push(MyData data)
3  {
4      MyData *pData = NULL;
5      pData = new MyData(data.data);      //生成新节点
6      pData->next = top;                  //与原来的栈顶节点相连
7      top = pData;                        //栈顶节点为新加入的节点
8  }
9
10  //出栈，返回栈顶节点内容
11  void MyStack::pop(MyData *data)
12  {
```

```
13      if (IsEmpty())                   //如果栈为空，则直接返回
14      {
15          return;
16      }
17      data->data = top->data;              //给传出的参数赋值
18      MyData *p = top;                 //临时保存原栈顶节点
19      top = top->next;                 //移动栈顶，指向下一个节点
20      delete p;                        //释放原栈顶节点内存
21  }
22
23  //判断栈是否为空栈
24  bool MyStack::IsEmpty()
25  {
26      return (top == NULL);              //如果top为空，则返回1，否
则返回0
27  }
```

MyStack::push 函数表示进栈操作，它实际上就是在单链表的首部进行插入操作，并且top一直指向这个单链表的首部。

MyStack::pop 函数表示出栈操作，它实际上就是在单链表的首部进行删除操作，并且top一直指向这个单链表的首部。另外，它还把删除节点的数据保存到data参数中（代码第17行）。

MyStack::IsEmpty函数非常简单。当空栈时，top指针为NULL；而当栈中有节点时， top指向链表头，因此只需要用top与NULL进行比较即可。

下面是栈操作的测试代码。

```
1   int main()
2   {
```

```
3    MyData data(0);                          //定义一个节点
4    MyStack s;                              //定义一个栈结构
5    s.push(MyData(1));                       //进栈三次: 1, 2, 3
6    s.push(MyData(2));
7    s.push(MyData(3));
8    s.pop(&data);                            //第1次出栈
9    cout << "pop " << data.data << endl;
10    s.pop(&data);                           //第2次出栈
11    cout <<    "pop " << data.data << endl;
12    s.pop(&data);                           //第3次出栈
13    cout <<    "pop " << data.data << endl;
14    cout << "Empty = " << s.IsEmpty() << endl;    //打印判空
15    return 0;
16    }
```

执行结果为：

```
1    pop 3
2    pop 2
3    pop 1
4    IsEmpty = 1
```

可以看出，进栈的顺序和出栈的顺序相反。

在前面的小节里，我们已经实现过queue，当时我们用了front和rear两个指针分别指向队头和队尾。这里由于题目限制，我们不能使用这些指针。

如何只使用stack实现queue呢？我们知道stack是先进后出的（FILO），而queue是先进先出的（FIFO）。也就是说，stack 进行了一次反向。如果进行两次反向，就能实现queue的功能，所以我们需要两个stack实现queue。

下面是具体的思路。

假设有两个栈A和B，且都为空。可以认为栈A为提供入队列的功能，栈B提供出队列的功能。

（1）如果栈B不为空，直接弹出栈B的数据。

（2）如果栈B为空，则依次弹出栈A的数据，放入栈B中，再弹出栈B的数据。

于是，我们可以得到下面的MyQueue定义及实现。

```
1   class MyQueue
2   {
3   public:
4       void enqueue(MyData data);          //入队
5       void dequeue(MyData &data);         //出队
6       bool IsEmpty();                     //是否为空队
7   private:
8       MyStack s1;                         //用于入队
9       MyStack s2;                         //用于出队
10   };
11
12   //入队
13   void MyQueue::enqueue(MyData data)
14   {
15       s1.push(data);                     //只对s1进行进栈操作
16   }
17
18   //出队
19   void MyQueue::dequeue(MyData &data)
20   {
```

21   MyData temp(0);   //局部变量，用于临时存储

22

23   if (s2.IsEmpty())

24   {

25   //如果s2为空，把s1的所有元素push到s2中

26   while(!s1.IsEmpty())

27   {

28   s1.pop(temp);   //弹出s1的元素

29   s2.push(temp);   //压入s2中

30   }

31   }

32   if (!s2.IsEmpty())

33   {

34   //此时如果s2不为空，则弹出s2的栈顶元素

35   s2.pop(data);

36   }

37   }

38

39   //队列判空

40   bool MyQueue::IsEmpty()

41   {

42   //如果两个栈都为空，则返回1，否则返回0

43   return (s1.IsEmpty() && s2.IsEmpty());

44   }

测试MyQueue的程序如下。

```
1   int main()
2   {
3       /*测试队列*/
4       MyData data(0);          //定义一个节点
5       MyQueue q;
6
7       q.enqueue(MyData(1));
8       q.enqueue(MyData(2));
9       q.enqueue(MyData(3));
10
11      q.dequeue(data);
12      cout << "dequeue " << data.data << endl;
13      q.dequeue(data);
14      cout << "dequeue " << data.data << endl;
15      q.dequeue(data);
16      cout << "dequeue " << data.data << endl;
17      cout << "IsEmpty: " << q.IsEmpty() << endl;
18
19      return 0;
20  }
```

执行结果：

```
1   dequeue 1
2   dequeue 2
3   dequeue 3
4   IsEmpty: 1
```

通过结果说明，入队顺序与出队顺序相同。

面试题26 选择题—栈的使用

考点：队列和栈的区别

出现频率：★★★★

设栈的最大长度为3，入栈序列为1，2，3，4，5，6，则不可能得出的栈序列是：

A．1，2，3，4，5，6

B．2，1，3，4，5，6

C．3，4，2，1，5，6

D．4，3，2，1，5，6

【解析】

此题隐含了一个前提，就是任何时候都能进栈和出栈。

下面我们具体分析每一个选项。

选项A：顺序为1，2，3，4，5，6。很明显，如果每次有一个元素进栈，然后马上出栈，即1进栈，1出栈，2进栈，2出栈，如此进行就可以遵循这个顺序。

选项B：顺序为2，1，3，4，5，6。先入栈1，2，然后全部出栈，即2，1出栈。接着后面的元素和选项 A 的方式一样，有一个元素进栈，然后马上出栈，这样也可以遵循这个顺序。

选项C：顺序为3，4，2，1，5，6。先入栈1，2，3，然后做一次出栈，即3出栈。接着4进栈，并且马上全部出栈，即4，2，1出栈。后面的5，6的入栈、出栈方式和选项A的一样，这样也可以遵循这个顺序。

选项D：顺序为4，3，2，1，5，6。注意，这里的前面四个元素是4，3，2，1。如果栈的最大长度为4，是可以实现按这个顺序出栈的。然而栈的最大长度为3，即元素4无法和1，2，3同时存在栈内。所以无

法按4，3，2，1顺序出栈。

【答案】

D

面试题27 用C++实现一个二叉排序树

用C++实现一个二叉排序树，完成创建、插入节点，删除节点，查找节点功能。

考点：二叉排序树的各项基本操作

出现频率：★★★★

【解析】

二叉排序树（Binary Sort Tree）又称二叉查找树（Binary Search Tree）。其定义为：二叉排序树或者是空树，或者是满足如下性质的二叉树。

（1）若它的左子树非空，则左子树上所有节点的值均小于根节点的值。

（2）若它的右子树非空，则右子树上所有节点的值均大于根节点的值。

（3）左、右子树本身又各是一棵二叉排序树。

上述性质简称二叉排序树性质（BST性质），故二叉排序树实际上是满足BST性质的二叉树。

图8.6就是一个简单的二叉排序树。

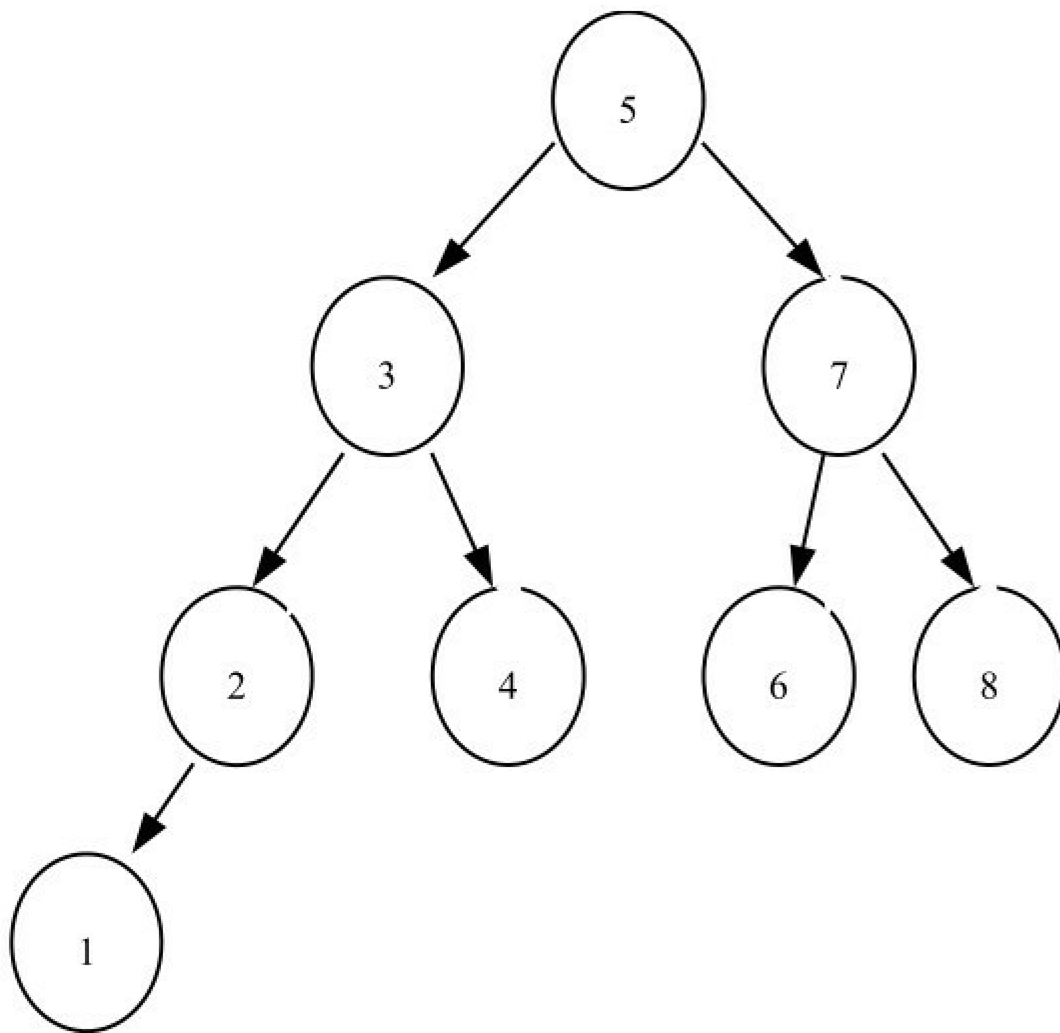


图8.6 二叉排序树模型

首先我们需要定义节点类以及二叉排序树类。对于节点类来说，显然每个节点有向下的两个指针，而且每个节点都只有一个父节点。对于二叉排序树类来说，只需要有数的根节点，就可以进行操作了。因此它们的数据结构如下。

```
1   //节点类定义
2   class Node
3   {
4   public:
5       int    data;                        //数据
```

```
6      Node *parent;                        //父亲节点
7      Node *left;                          //左子节点
8      Node *right;                         //右子节点
9    public:
10       Node() : data(-1), parent(NULL), left(NULL), right(NULL) { };
11       Node(int num) : data(num), parent(NULL), left(NULL),
right(NULL) { };
12   };
13
14   //二叉排序树类定义
15   class Tree
16   {
17   public:
18       Tree(int num[], int len);                  //插入num数组的前len个
数据
19       void insertNode1(int data);                //插入节点，递归方法
20       void insertNode(int data);                 //插入节点，非递归方法
21       Node *searchNode(int data);                //查找节点
22       void deleteNode(int data);                 //删除节点及其子树
23   private:
24       void insertNode(Node* current,int data);       //递归插入方法
25       Node *searchNode(Node* current,int data);      //递归查找方法
26       void deleteNode(Node* current);                //递归删除方法
27   private:
28       Node* root;                                //二叉排序树的根节点
29   };
```

第18行的构造函数定义了根据int数组创建二叉排序树的方法。

第19～20行定义两种插入节点的方法，分别为递归和非递归方法。

第21～22行分别定义了查找和删除节点的方法，这两个方法均为递归方法。

另外注意，第24～26行中有3个private方法。这是因为，出于封装性的考虑，指针root是私有成员变量，于是3个使用递归方法的函数（代码第20～22行）都只能有一个参数data。然而，这里只含有一个参数data的函数是无法进行递归运算的。因此它们内部直接调用了这3个private的对应方法，而这3个private方法直接使用递归。

接下来具体说明各个方法的实现。

1．创建二叉排序树的方法（构造函数中）

这里可以首先生成根节点，然后循环调用插入节点的方法对二叉树进行插入操作。

```
1   //插入num数组的前len个数据
2   Tree::Tree(int num[], int len)
3   {
4       root = new Node(num[0]);        //建立root节点
5       //把数组中的其他数组插入到二叉排序树中
6       for(int i=1; i < len; i++)
7       {
8           //insertNode(num[i]);
9           insertNode1(num[i]);
10      }
11  };
```

2．插入节点操作

通常在二叉树的遍历中可以选择递归与非递归的方法。由于篇幅有限，本题中仅给出插入时的这两种方法。

insertNode1()使用非递归方法插入节点，其代码如下。

```
1    //插入数据为参数data的节点,非递归方法
2    void Tree::insertNode1(int data)              //插入节点
3    {
4       Node *p, *par;
5       Node *newNode = new Node(data);           //创建节点
6
7       p = par = root;
8       while(p != NULL)                          //查找插入在哪个节点下面
9       {
10          par = p;    //保存节点
11          if (data > p->data)                   //如果data大于当前节点的data
12              p = p->right;                     //下一步到左子节点,否则进行
到右子节点
13          else if(data < p->data)
14              p = p->left;
15          else if (data == p->data)             //不能插入重复数据
16          {
17             delete newNode;
18             return;
19          }
20       }
21       newNode->parent = par;
22       if (par->data > newNode->data)           //把新节点插入在目标
节点的正确位置
23          par->left = newNode;
24       else
25          par->right = newNode;
```

```
26   }
```

它分为下面的步骤。

（1）创建节点（代码第5行）。

（2）查找新建节点的插入位置。

根据前面介绍过的排序二叉树的性质，我们进行节点的比较（代码第11～14行）进行节点的遍历。如果二叉树中已经含有相同数据的节点，则不予插入（代码第15～19行）且直接返回。循环完毕后，p为NULL，par指向目标节点。

（3）把新节点插入在目标节点的正确位置。与（2）相同，需要考虑到排序二叉树的性质。

insertNode()使用非递归的方法插入节点，它的内部调用了private函数insertNode()。代码如下。

```
1   //插入数据为参数data的节点,调用递归插入方法
2   void Tree::insertNode(int data)
3   {
4       if(root != NULL)
5       {
6           insertNode(root, data);                //调用递归插入方法
7       }
8   }
9
10  //递归插入方法
11  void Tree::insertNode(Node* current,int data)
12  {
13      //如果data小于当前节点数据，则在当前节点的左子树插入
14      if(data < current->data)
15      {
```

```
16          if(current->left == NULL)              //如果左节点不存在，则
插入到左节点
17          {
18                current->left = new Node(data);
19                current->left->parent = current;
20          }
21          else
22                insertNode(current->left,data);   //否则对左节点进行递
归调用
23       }
24
25       //如果data大于当前节点数据，则在当前节点的右子树插入
26       else if(data > current->data)
27       {
28          if(current->right == NULL)              //如果右节点不存在，
则插入到右节点
29          {
30                current->right = new Node(data);
31                current->right->parent = current;
32          }
33          else
34                insertNode(current->right,data);   //否则对右节点进行
递归调用
35       }
36
37       return;                                //data等于当前节点数据时，不
插入
```

38　};

现在说明private成员函数insertNode的步骤。

（1）如果当前节点的数据值小于data，则应该在它的左子树插入。此时如果当前节点没有左子节点，则直接插入新节点作为它的左子节点。否则把它的左子节点作为参数，再进行整个过程。

（2）如果当前节点的数据值大于data，则应该在它的右子树插入。此时如果当前节点没有右子节点，则直接插入新节点作为它的右子节点。否则把它的右子节点作为参数，再进行整个过程。

比较这两个版本的插入函数可以看出，使用递归方法写出的程序简单，容易理解；而非递归方法使用循环，与递归方法的程序结构相比显得臃肿。递归时需要不断地进行函数入栈、出栈操作，因此效率方面较低；并且如果递归的深度较大，可能会引发栈溢出。

查找节点也使用了递归，由于与插入节点类似，这里只列出private方法。

```
1   //递归查找方法
2   Node* Tree::searchNode(Node* current,int data)
3   {
4     //如果data小于当前节点数据，则递归搜索其左子树
5     if(data < current->data)
6     {
7       if (current->left == NULL)          //如果不存在左子树，则返回NULL
8           return NULL;
9       return searchNode(current->left, data);
10     }
11
12     //如果data大于当前节点数据，则递归搜索其左子树
```

```
13      else if(data > current->data)
14      {
15         if (current->right == NULL)      //如果不存在右子树，则返回NULL
16               return NULL;
17         return searchNode(current->right, data);
18      }
19
20      //如果相等，则返回current
21      return current;
22   }
```

searchNode的步骤如下。

（1）如果data小于当前节点（current）的值，且current的左子树存在，则继续搜索current的左子树，否则返回NULL。

（2）如果data大于当前节点（current）的值，且current的右子树存在，则继续搜索current的左子树，否则返回NULL。

（3）如果data等于当前节点（current）的值，则返回current。

3．删除节点

删除节点的操作代码如下。

```
1   //删除数据为data的节点及其子树
2   void Tree::deleteNode(int data)
3   {
4      Node *current = NULL;
5
6      current = searchNode(data);      //查找节点
7      if (current != NULL)
8      {
```

```
9           deleteNode(current);       //删除节点及其子树
10        }
11    }
12
13    //删除current节点及其子树的所有节点
14    void Tree::deleteNode(Node *current)
15    {
16        if (current->left != NULL)       //删除左子树
17            deleteNode(current->left);
18        if (current->right != NULL)       //删除右子树
19            deleteNode(current->right);
20
21        if (current->parent == NULL)
22        {
23            //如果current是根节点，把root置空
24            delete current;
25            root = NULL;
26            return;
27        }
28
29        //将current父亲节点的相应指针置空
30        if (current->parent->data > current->data) //current为其父节点
的左子节点
31                current->parent->left = NULL;
32        else    //current为parNode的右子节点
33            current->parent->right = NULL;
34
```

```
35      //最后删此节点
36      delete current;
37  }
```

public 成员函数 deleteNode()删除数据为 data 的节点及其子树。它首先调用了searchNode()查找数据等于data的节点（代码第6行），如果找到节点，则调用private成员函数deleteNode的节点及其子树。

private成员函数也是使用递归方法进行删除操作的。它的步骤如下。

（1）如果current左子树存在，则递归删除current左子树。

（2）如果current右子树存在，则递归删除current右子树。

（3）最后删除current节点。此时如果current是根节点，则需要把root置空，否则把其父节点相应的指针置空。

面试题28 使用递归与非递归方法实现中序遍历

考点：中序遍历算法的实现

出现频率：★★★

【解析】

中序遍历的递归算法定义为，若二叉树非空，则依次执行如下操作。

（1）遍历左子树；

（2）访问根节点；

（3）遍历右子树。

中序遍历的递归算法程序代码如下。

```
1   //中序遍历
2   void Tree::InOrderTree()
3   {
4       if(root == NULL)
5               return;
6       InOrderTree(root);
7   }
8
9   void Tree::InOrderTree(Node* current)
10  {
11      if(current!= NULL)
12      {
13          InOrderTree(current->left);       //遍历左子树
```

```
14          cout << current->data << " ";        //打印节点数据
15          InOrderTree(current->right);        //遍历右子树
16      }
17  }
```

对于中序遍历非递归算法，这里使用一个栈（stack）来临时存储节点，方法如下。

（1）先将根节点入栈，遍历左子树。

（2）遍历完左子树返回时，栈顶元素应为根节点，此时出栈，并打印节点数据。

（3）再中序遍历右子树。

代码如下。

```
1   void Tree::InOrderTreeUnRec()
2   {
3       stack<Node *> s;
4       Node *p = root;
5       while(p != NULL || !s.empty())
6       {
7           while(p != NULL)                //遍历左子树
8           {
9                   s.push(p);              //把遍历的节点全部压栈
10                p = p->left;
11          }
12
13          if (!s.empty())
14          {
15                p = s.top();          //得到栈顶内容
16                s.pop();              //出栈
```

17　　　　　cout << p->data << " ";　//打印

18　　　　　p = p->right;　　　//指向右子节点，下一次循环时就会中序遍历右子树

19　　　}

20　　}

21　}

main函数测试如下。

```
1   int main(void)
2   {
3       int num[] = {5, 3 ,7, 2, 4, 6, 8, 1};
4       Tree tree(num, 8);
5       cout << "InOrder: ";
6       tree.InOrderTree();             //中序遍历，递归方法
7       cout << "\nInOrder: ";
8       tree.InOrderTreeUnRec();        //中序遍历，非递归方法
9       return 0;
10  }
```

测试结果为

```
1   1 2 3 4 5 6 7 8
2   1 2 3 4 5 6 7 8
```

另外，从这个结果可以看出，中序遍历的结果就是二叉排序树的排序结果。

面试题29 使用递归与非递归方法实现先序遍历

考点：先序遍历算法的实现

出现频率：★★★

【解析】

先序遍历的递归算法定义为，若二叉树非空，则依次执行如下操作。

（1）访问根结点；

（2）遍历左子树；

（3）遍历右子树。

先序遍历的程序代码如下。

```
1    //先序遍历
2    void Tree::PreOrderTree()
3    {
4        if(root == NULL)
5                return;
6        PreOrderTree(root);
7    };
8
9    void Tree::PreOrderTree(Node* current)
10    {
11        if(current!= NULL)
12        {
13            cout << current->data << " ";        //打印节点数据
```

```
14          PreOrderTree(current->left);       //遍历左子树
15          PreOrderTree(current->right);      //遍历右子树
16      }
17      }
```

对于先序遍历非递归算法，这里我们使用一个栈（stack）来临时存储节点，方法如下。

（1）打印根节点数据。

（2）把根节点的right入栈，遍历左子树。

（3）遍历完左子树返回时，栈顶元素应为right，出栈，遍历以该指针为根的子树。

程序如下

```
1   void Tree::PreOrderTreeUnRec()
2   {
3     stack<Node *> s;
4     Node *p = root;
5     while(p != NULL || !s.empty())
6     {
7       while(p != NULL)              //遍历左子树
8       {
9           cout << p->data << " ";   //打印
10          s.push(p);               //把遍历的节点全部压栈
11          p = p->left;
12      }
13
14      if (!s.empty())
15      {
16          p = s.top();          //得到栈顶内容
```

```
17                s.pop();              //出栈
18                p = p->right;              //指向右子节点，下一次循环时就会先序遍历左子树
19          }
20       }
21    }
```

main函数测试如下。

```
1   int main(void)
2   {
3       int num[] = {5, 3 ,7, 2, 4, 6, 8, 1};
4       Tree tree(num, 8);
5       cout << "PreOrder: ";
6       tree.PreOrderTree();              //先序遍历，递归方法
7       cout << "\nPreOrder: ";
8       tree.PreOrderTreeUnRec();         //先序遍历，非递归方法
9       return 0;
10    }
```

测试结果为：

```
1   5 3 2 1 4 7 6 8
2   5 3 2 1 4 7 6 8
```

面试题30 使用递归与非递归方法实现后序遍历

考点：后序遍历算法的实现

出现频率：★★★

【解析】

后序遍历的递归算法定义为，若二叉树非空，则依次执行如下操作。

（1）遍历左子树；

（2）遍历右子树；

（3）访问根结点。

后序遍历的程序代码如下。

```
1   //后序遍历
2   void Tree::PostOrderTree()
3   {
4       if(root == NULL)
5               return;
6       PostOrderTree(root);
7   }
8
9   void Tree::PostOrderTree(Node* current)
10   {
11       if(current!= NULL)
12       {
13           PostOrderTree(current->left);       //遍历左子树
```

```
14          PostOrderTree(current->right);       //遍历右子树
15          cout << current->data << " ";       //打印节点数据
16       }
17   }
```

现在说明对于后序遍历的非递归方法。假设 T 是要遍历树的根指针，后序遍历要求在遍历完左、右子树后再访问根。需要判断根结点的左、右子树是否均遍历过。

可采用标记法，结点入栈时，配一个标志tag一同入栈（tag为0表示遍历左子树前的现场保护，tag为1表示遍历右子树前的现场保护）。

首先将T和tag（为0）入栈，遍历左子树；返回后，修改栈顶tag为1，遍历右子树；最后访问根结点。代码如下。

```
1   void Tree::PostOrderTreeUnRec()
2   {
3      stack<Node *> s;
4      Node *p = root;
5
6      while(p != NULL || !s.empty())
7      {
8         while(p != NULL)
9         {
10            s.push(p);              //压栈
11            p = p->left;              //遍历左子树
12         }
13
14         if (!s.empty())
15         {
16            p = s.top();          //得到栈顶元素
```

```
17                  if (p->tag)              //tag为1时
18                  {
19                          cout << p->data << " ";    //打印节点数据
20                          s.pop();           //出栈
21                          p = NULL;       //第二次访问标志其右子树已经遍历
22              }
23              else
24              {
25                          p->tag = 1;       //修改tag为1
26                          p = p->right;    //指向右节点，下次遍历其左子树
27              }
28          }
29      }
30  }
```

根据上面的分析，代码中Node类添加了一个 int型的成员变量 tag。main函数测试如下。

```
1   int main(void)
2   {
3       int num[] = {5, 3 ,7, 2, 4, 6, 8, 1};
4       Tree tree(num, 8);
5       cout << "PostOrder: ";
6       tree.PostOrderTree();           //后序遍历，递归方法
7       cout << "\nPostOrder: ";
8       tree.PostOrderTreeUnRec();       //后序遍历，非递归方法
9       return 0;
10  }
```

测试结果为：

1　1 2 4 3 6 8 7 5

2　1 2 4 3 6 8 7 5

面试题31 编写层次遍历二叉树的算法

考点：层次遍历算法的实现

出现频率：★★

【解析】

层次遍历就是一层一层地进行遍历。比如图8.6共有4层，各层分别为

第1层：5

第2层：3、7

第3层：2、4、6、8

第4层：1

本题很难直接用节点的指针（left、right、parent）来实现，但是借助一个队列就可以轻松地实现，例如图8.7所示的二叉树结构。

可以按照下面的方式执行。

（1）A入队CX。

（2）A出队，同时A的子节点B、C入队（此时队列有B、C）。

（3）B出队，同时B的子节点D、E入队（此时队列有C、D、E）。

（4）C出队，同时C的子节点F、G入队（此时队列有D、E、F、G）。

显然，这样就能使出队的顺序满足层次遍历。

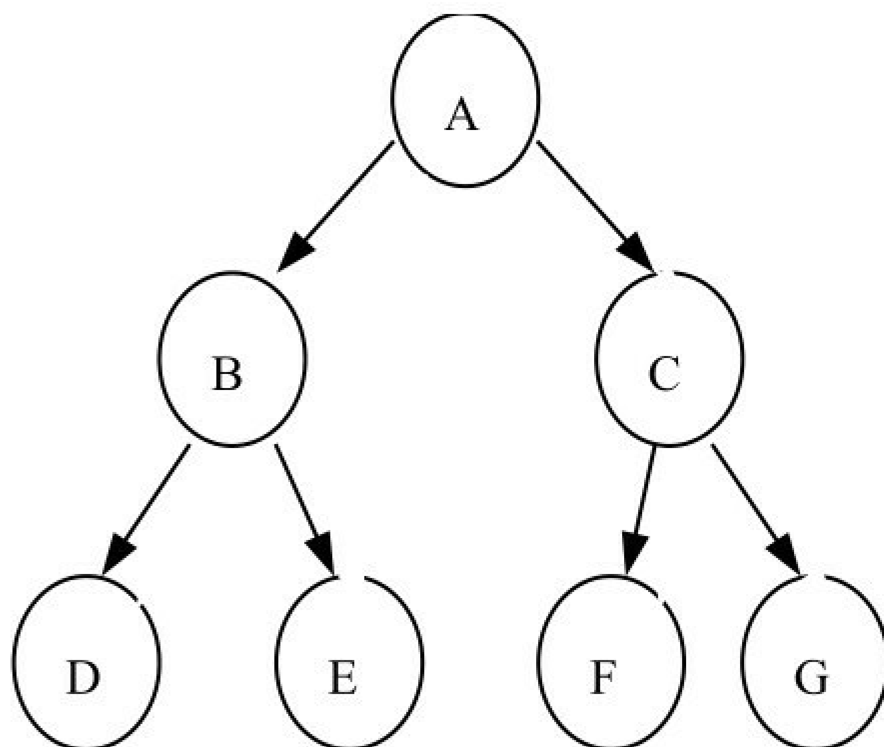


图8.7 二叉树图

这里为了方便，我们选择使用C++标准库中的queue模板类。代码如下。

```
1   //层次遍历
2   void Tree::LevelOrderTree()
3   {
4       queue<Node *> q;                    //定义队列q，它的元素为Node *类型指针
5       Node *ptr = NULL;
6
7       q.push(root);                       //根节点入队
8       while(!q.empty())
9       {
10          ptr = q.front();                //得到队头节点
```

```
11          q.pop();              //出队
12          cout << ptr->data << " ";       //当前节点存在左节点，则左节点入队
13          {
14              q.push(ptr->left);
15          }
16          if (ptr->right != NULL)       //当前节点存在右节点，则右节点入队
17          {
18              q.push(ptr->right);
19          }
20      }
21  }
```

程序中对队列 q 进行入队和出队的操作。每次出队时，就打印此出队的元素，并且对这个元素节点的左子节点和右子节点进行入队。

测试代码如下。

```
1   int main(void)
2   {
3       int num[] = {5, 3 ,7, 2, 4, 6, 8, 1};
4       Tree tree(num, 8);
5       cout << "InOrder: ";
6       tree.LevelOrderTree();       //层次遍历
7       return 0;
8   }
```

执行结果如下。

```
1   5 3 7 2 4 6 8 1
```

面试题32 编写判别给定二叉树是否为二叉排序树的算法

考点：二叉排序树判定算法的实现

出现频率：★★★

【解析】

在前面的面试题中，我们已经编写过中序遍历的算法。我们知道，使用中序遍历的结果就是排序二叉树的排序输出，因此我们可以使用中序遍历来实现判定二叉树是否为二叉排序树。代码如下：

```
1    //使用中序遍历判断二叉树是否为排序二叉树
2    bool IsSortedTree(Tree tree)
3    {
4        int lastvalue = 0;
5        stack<Node *> s;
6        Node *p = tree.root;
7        while(p != NULL || !s.empty())
8        {
9            while(p != NULL)          //遍历左子树
10           {
11                s.push(p);          //把遍历的节点全部压栈
12                p = p->left;
13           }
14
15           if (!s.empty())
16           {
```

17   p = s.top();   //得到栈顶内容

18   s.pop();   //出栈

19   if (lastvalue == 0 || lastvalue < p->data)

20   {

21   //如果是第一次弹出或者lastvalue小于当前节点值，给lastvalue赋值

22   lastvalue = p->data;

23   }

24   else if (lastvalue >= p->data)

25   {

26   //如果lastvalue大于当前节点值,则返回false

27   return false;

28   }

29

30   p = p->right;   //指向右子节点，下一次循环时就会中序遍历右子树

31   }

32   }

33

34   return true;   //到这里说明节点数据是升序排列，返回true

35   }

这里我们使用了非递归的二叉树遍历方式，其中使用了一个lastvalue 来记录上一次出队的节点数据。如果出现lastvalue大于或等于当前出队的节点值，则返回false；否则一直入队和出队。如果lastvalue始终小于当前出队的节点值，则是升序排列，返回true。

下面是主函数的测试代码。

```
1    int main(void)
2    {
3        int num[] = {5, 3 ,7, 2, 4, 6, 8, 1};
4        Tree tree(num, 8);
5        cout << "InOrder: ";
6        tree.InOrderTreeUnRec();            //中序遍历，非递归方法
7        cout << "\nIsSortedTree: " << IsSortedTree(tree) << endl;
8        Node *node = tree.searchNode(4);
9        node->data = 1;                     //手动把节点的数据改成1
10       cout << "InOrder: ";
11       tree.InOrderTreeUnRec();            //中序遍历，非递归方法
12       cout << "\nIsSortedTree: " << IsSortedTree(tree) << endl;
13       return 0;
14   }
```

测试结果：

```
1    1 2 3 4 5 6 7 8
2    IsSortedTree: 1
3    1 2 3 1 5 6 7 8
4    IsSortedTree: 0
```

第9章 排序

所谓排序，就是要整理数组中的记录，使之按关键字递增（或递减）的次序排列起来。其确切定义如下。

输入：n个记录R_1，R_2，…，R_n，其相应的关键字分别为K_1，K_2，…，K_n。

输出：R_{il}，R_{i2}，…，R_{in}，使得$K_{i1} \leq K_{i2} \leq \ldots \leq K_{in}$（或$K_{i1} \geq K_{i2} \geq \ldots \geq K_{in}$）。

排序可以分为下面5类。

插入排序；

选择排序；

交换排序；

归并排序；

分配排序。

面试题1 编程实现直接插入排序

考点：直接插入排序算法的实现

出现频率：★★★★

【解析】

1．直接插入排序

直接插入排序是稳定的排序方法。直接插入排序的基本思想：假设待排序的记录存放在数组R[1…n]中。初始时，R[1]自成1个有序区，无序区为R[2…n]。从i=2起直至i=n为止，依次将R[i]插入当前的有序区R[1...i-1]中，生成含n个记录的有序区。

第i-1趟直接插入排序：

通常将一个记录R[i]（i=2，…，n-1）插入到当前的有序区，使得插入后仍保证该区间里的记录是按关键字有序的操作，称为第i-1趟直接插入排序。

排序过程的某一中间时刻，R被划分成两个子区间：R[1…i-1]（已排好序的有序区）和R[i…n]（当前未排序的部分，可称为无序区）。

直接插入排序的基本操作是将当前无序区的第1个记录R[i]插入到有序区R[1…i-1]中适当的位置上，使R[1…i]变为新的有序区。因为这种方法每次使有序区增加1个记录，通常称为增量法。

插入排序与打扑克时整理手上的牌非常类似。摸来的第 1 张牌无须整理，此后每次从桌上的牌（无序区）中摸最上面的 1 张并插入左手的牌（有序区）中正确的位置上。为了找到这个正确的位置，须自左向右（或自右向左）将摸来的牌与左手中已有的牌逐一比较。

由直接插入排序的基本思想很容易得到下面简单的方法。

（1）在当前有序区R[1…i-1]中查找R[i]的正确插入位置k（1≤k≤i-

1）。

（2）将R[k…i-1]中的记录均后移一个位置，腾出k位置上的空间插入R[i]。

这里我们使用升序排序，也就是说，如果 R[i]的关键字大于等于R[1…i-1]中所有记录的关键字，则R[i]就是插入的位置。

还有一种改进的方法，即查找比较操作和记录移动操作交替地进行。其具体做法如下。

将待插入记录R[i]的关键字从右向左依次与有序区中记录R[j]（j=i-1，i-2，…，1）的关键字进行比较：

（1）如果R[j]的关键字大于R[i]的关键字，则将R[j]后移一个位置;

（2）如果R[j]的关键字小于或等于R[i]的关键字，则查找过程结束，j+1即为R[i]的插入位置。

关键字比R[i]的关键字大的记录均已后移，所以j+1的位置已经腾空，只要将R[i]直接插入此位置即可完成一趟直接插入排序。

2．实现

我们使用上面介绍的改进的方法，即查找比较操作和记录移动操作交替地进行。代码如下。

```
1    #include <iostream>
2    using namespace std;
3
4    //直接插入排序
5    void insert_sort(int a[], int n)
6    {
7        int i, j, temp;
8
9        for (i=1; i<n; i++) //需要选择n-1次
```

```
10      {
11          //暂存下标为i的数。下标从1开始，因为开始时
12          //下标为0的数，前面没有任何数，此时认为它是排好顺
序的
13          temp = a[i];
14          for (j=i-1; j>=0 && temp<a[j]; j--)
15          {
16              //如果满足条件就往后挪。最坏的情况就是temp比a[0]
小，它要放在最前面
17              a[j+1] = a[j];
18          }
19
20          a[j+1] = temp;          //找到下标为i的数的放置位置
21      }
22  }
23
24  void print_array(int a[], int len)
25  {
26      for(int i = 0; i < len; i++)      //循环打印数组的每个元素
27      {
28          cout << a[i] << " ";
29      }
30      cout << endl;
31  }
32
33  int main()
34  {
```

```
35        int a[] = {7, 3, 5, 8, 9, 1, 2, 4, 6};
36        cout << "before insert sort: ";
37        print_array(a, 9);
38        insert_sort(a, 9);                //进行直接插入排序
39        cout << "after insert sort: "
40        print_array(a, 9);
41        return 0;
42    }
```

insert_sort函数的插入次数是len-1，因为当数组只有一个a[0]时，我们认为a[0]就是已经排好序的了。局部变量i用于表示对哪一个元素进行插入操作，j表示插入到哪个目标元素的后面，temp保存需要插入的元素。这里最坏的情况就是temp比a[0]都小，此时j为-1，需要把temp作为新的a[0]。

测试结果如下。

```
1    before insert sort: 7 3 5 8 9 1 2 4 6
2    before insert sort: 1 2 3 4 5 6 7 8 9
```

面试题2 编程实现希尔（Shell）排序

考点：Shell排序算法的实现

出现频率：★★★★

【解析】

1．希尔（Shell）排序

希尔（Shell）排序是D.L.shell于1959年提出的，它属于插入排序方法，是不稳定的排序方法。

我们知道，在直接插入排序算法中，每次插入一个数，使有序序列只增加 1 个节点，并且对插入下一个数没有提供任何帮助。如果比较相隔较远距离（称为增量）的数，使得数移动时能跨过多个元素，则进行一次比较就可能消除多个元素交换。

希尔（Shell）排序算法先将要排序的一组数按某个增量 d 分成若干组，每组中记录的下标相差 d 对每组中全部元素进行排序，然后用一个较小的增量对它进行再次分组，并对每个新组重新进行排序。当增量减到 1 时，整个要排序的数被分成一组，排序完成。因此希尔排序实质上是一种分组插入方法。

希尔排序的时间性能优于直接插入排序，其原因如下。

当数组初始状态基本有序时，直接插入排序所需的比较和移动次数均较少。

当n 值较小时，n 和n2 的差别也较小，即直接插入排序的最好时间复杂度O(n)和最坏时间复杂度0(n2)差别不大。

在希尔排序开始时，增量较大，分组较多，每组的记录数目少，故各组内直接插入较快，后来增量 d 逐渐缩小，分组数逐渐减少，而各组的记录数目逐渐增多。但由于已经按d-1作为距离排过序，数组较接近

于有序状态，所以新的一趟排序过程也较快。

因此，希尔排序在效率上较直接插入排序有较大的改进。

另外，由于分组的存在，相等的元素可能会分在不同组，导致它们的次序可能发生变化，因此希尔排序是不稳定的。

2．实现

我们可以这样来设置增量：初始时取序列的一半为增量，以后每次减半，直到增量为1。

代码如下。

```
1   #include <iostream>
2   using namespace std;
3
4   void shell_sort(int a[], int len)
5   {
6       int h, i, j, temp;
7
8       for (h=len/2; h>0; h=h/2)            //控制增量
9       {
10          for (i=h; i<len; i++)            //这个for循环就是前面的直接
插入排序
11          {
12              temp = a[i];
13              for (j=i-h; (j>=0 && temp<a[j]); j-=h)
14              {
15                  a[j+h] = a[j];
16              }
17              a[j+h] = temp;
18          }
```

```
19      }
20  }
21
22  void print_array(int a[], int len)
23  {
24      for(int i = 0; i < len; i++)            //循环打印数组的每个元素
25      {
26          cout << a[i] << " ";
27      }
28      cout << endl;
29  }
30
31  int main()
32  {
33      int a[] = {7, 3, 5, 8, 9, 1, 2, 4, 6};
34      cout << "before shell sort: ";
35      print_array(a, 9);
36      shell_sort(a, 9);                   //进行Shell排序
37      cout << "after shell sort: ";
38      print_array(a, 9);
39      return 0;
40  }
```

shell_sort函数使用了循环设置增量（代码第8行），里面又嵌套了一个直接插入排序的算法。注意这个嵌套的算法代码实现，它与上个例题中的代码相比只有一点不同，就是现在的增量是h，而原来的增量是1。

测试结果如下。

```
1    before shell sort: 7 3 5 8 9 1 2 4 6
2    before shell sort: 1 2 3 4 5 6 7 8 9
```

面试题3 编程实现冒泡排序

考点：冒泡排序算法的实现

出现频率：★★★★

【解析】

1．冒泡排序

冒泡排序的方法为：将被排序的记录数组A[1...n]垂直排列，每个记录A[i]看作重量为A[i]气泡。根据轻气泡不能在重气泡之下的原则，从下往上扫描数组A：凡扫描到违反本原则的轻气泡，就使其向上“飘浮”。如此反复进行，直到最后任何两个气泡都是轻者在上、重者在下为止。

冒泡排序是稳定的排序。下面是具体的算法。

（1）初始状态下，A[1…n]为无序区。

（2）第一趟扫描：从无序区底部向上依次比较相邻的两个气泡的重量，若发现轻者在下、重者在上，则交换二者的位置。即依次比较（A[n]，A[n-1]），（A[n-1]，A[n-2]），…， （A[2]，A[1]）；对于每对气泡（A[j+1]，A[j]），若A[j+1]<A[j]，则交换A[j+1]和A[j]的内容。

第一趟扫描完毕时，“最轻”的气泡就飘浮到该区间的顶部，即关键字最小的记录被放在最高位置A[1]上。

（3）第二趟扫描：扫描A[2…n]。扫描完毕时，“次轻”的气泡飘浮到A[2]的位置上。

（4）第i趟扫描：A[1...i-1]和A[i…n]分别为当前的有序区和无序区。扫描仍是从无序区底部向上，直至该区顶部。扫描完毕时，该区中最轻气泡飘浮到顶部位置A[i]上，结果是A[1...i]变为新的有序区。

最后，经过n-1趟扫描可得到有序区A[1...n]。

2．实现

根据前面冒泡扫描的方法，可以写出下面的排序代码。

```
1   void bubble_sort_1(int a[], int len)
2   {
3       int i = 0;
4       int j = 0;
5       int temp = 0;                       //用于交换
6
7       for(i=0; i<len-1; i++)                  //进行n-1趟扫描
8       {
9           for(j=len-1; j>=i; j--)            //从后往前交换，这样最小值
冒泡到开头部分
10          {
11              if(a[j+1] < a[j])          //如果a[j]小于a[j-1]，则交换两
元素的值
12              {
13                  temp = a[j];
14                  a[j] = a[j+1];
15                  a[j+1] = temp;
16              }
17          }
18      }
19  }
```

这个代码有一个小问题，就是假如进行第i次扫描前，数组已经排好序了，但是它还会进行下一次的扫描，显然以后的扫描都是没有必要的。

我们可以对上面的这个代码进行一点改进，代码如下。

```
1    void bubble_sort_2(int a[], int len)
2    {
3        int i = 0;
4        int j = 0;
5        int temp = 0;                   //用于交换
6        int exchange = 0;                   //用于记录每次扫描时是否发生交换
7
8        for(i=0; i<len-1; i++)              //进行n-1趟扫描
9        {
10           exchange = 0;                  //每趟扫描之前对exchange置0
11           for(j=len-1; j>=i; j--)        //从后往前交换，这样最小值冒泡到开头部分
12           {
13               if(a[j+1] < a[j])       //如果a[j]小于a[j-1],交换两元素的值
14               {
15                   temp = a[j];
16                   a[j] = a[j+1];
17                   a[j+1] = temp;
18                   exchange = 1;       //发生交换，exchange置1
19               }
20           }
21           if (exchange != 1)          //此趟扫描没有发生过交换，说明已经是排序的
22               return;                  //不需要进行下次扫描
```

```
23      }
24  }
```

这里我们使用一个局部变量exchange来记录在本次扫描时有没有进行过数据交换。每次扫描之前，把exchange置0（代码第10行）。如果扫描时发生数据交换，则把exchange置1（代码第18行）；如果没有，则说明数组已经是排序的了，不需要进行下一趟扫描（代码第22行）。

对两种冒泡排序的测试main函数如下。

```
1   int main()
2   {
3       int a[] = {7, 3, 5, 8, 9, 1, 2, 4, 6};
4       cout << "before bubble sort: ";
5       print_array(a, 9);
6       //bubble_sort_1(a, 9);            //冒泡排序
7       bubble_sort_2(a, 9);              //改进的冒泡排序
8       cout << "after bubble sort: ";
9       print_array(a, 9);
10      return 0;
11  }
```

测试结果如下。

```
1   before bubble sort: 7 3 5 8 9 1 2 4 6
2   before bubble sort: 1 2 3 4 5 6 7 8 9
```

面试题4 编程实现快速排序

考点：快速排序算法的实现

出现频率：★★★★

【解析】

1．快速排序

快速排序是C.R.A.Hoare于1962年提出的一种划分交换排序。它采用了一种分治的策略，通常称其为分治法（Divide-and-ConquerMethod）。分治法的基本思想是：将原问题分解为若干个规模更小但结构与原问题相似的子问题。递归地解这些子问题，然后将这些子问题的解组合为原问题的解。

快速排序的基本思想：设当前待排序的无序区为A[low…high]，利用分治法可描述为：

（1）分解：在 A[low...high]中任选一个记录作为基准（pivot），以此基准将当前无序区划分为左、右两个较小的子区间A[low…pivotpos-1]和A[pivotpos+1…high]，并使左边子区间中所有记录的关键字均小于等于基准记录（pivot），右边的子区间中所有记录的关键字均大于等于pivot，而基准记录pivot则位于正确的位置上，它无须参加后续的排序。

注意，划分的关键是要求出基准记录所在的位置pivotpos。划分的结果可以简单地表示为（pivot=A[pivotpos]）：A[low…pivotpos-1]≤A[pivotpos]≤A[pivotpos+1…high]，其中low≤pivotpos≤high。

（2）求解：通过递归调用快速排序对左、右子区间A[low...pivotpos-1]和 A[pivotpos+1...high]快速排序。

（3）组合：当“求解”步骤中的两个递归调用结束时，其左、右两

个子区间已有序。对快速排序而言，“组合”步骤无须做什么，可看作空操作。

2．实现

源代码如下。

```
1   void quick_sort(int a[], int low, int high)
2   {
3       int i, j, pivot;
4       if (low < high)
5       {
6               pivot = a[low];
7               i = low;
8               j = high;
9               while(i<j)
10              {
11                  while (i<j && a[j]>=pivot)
12                      j--;
13                  if(i<j)
14                      a[i++]=a[j];          //将比pivot小的元素移到低端
15
16                  while (i<j && a[i]<=pivot)
17                      i++;
18                  if(i<j)
19                      a[j--] = a[i];          //将比pivot大的元素移到高端
20              }
21              a[i] = pivot;                   //pivot移到最终位置
22              quick_sort(a, low, i-1);            //对左区间递归排序
23              quick_sort(a, i+1, high);            //对右区间递归排序
```

```
24        }
25    }
```

这里pivot代表基准值，它的初始值为a[low]。局部变量i和j分别代表low和high的位置。接着按照下面的步骤进行一趟交换。

（1）把比pivot小的元素移到低端（low）。

（2）把比pivot大的元素移到高端（high）。

（3）pivot移到最终位置，此时这个位置的左边元素的值都比pivot小，而其右边元素的值都比pivot大。

（4）对左、右区间分别进行递归排序。从而把前三步的粗排序逐渐地细化，直至最终low和high交汇。

测试程序如下。

```
1    void main()
2    {
3        int data[9] = {54,38,96,23,15,72,60,45,83};
4        quick_sort(data, 0, 8);                    //快速排序
5        for(int i = 0; i<9; i++)
6                cout << data[i] << " ";            //打印排序后的数组
7    }
```

执行结果：

```
1    15 23 38 45 54 60 72 83 96
```

面试题5 编程实现选择排序

考点：直接选择排序算法的实现

出现频率：★★★

【解析】

1．直接选择排序

直接选择排序的基本思想：n个记录的直接选择排序可经过n-1趟直接选择排序得到有序结果。

（1）初始状态：无序区为A[1...n]，有序区为空。

（2）第1 趟排序：在无序区 A[1...n]中选出最小的记录 A[k]，将它与无序区的第 1个记录A[1]交换，使A[1…1]和A[2...n]分别变为记录个数增加1的新有序区和记录个数减少1的新无序区。

（3）第i趟排序：第i趟排序开始时，当前有序区和无序区分别为A[1…i-1]和A[i..n](1≤i≤n-1)。该趟排序从当前无序区中选出关键字最小的记录 A[k]，将它与无序区的第 1个记录A[i]交换，使A[1…i]和A[i+1...n]分别变为记录个数增加1的新有序区和记录个数减少1的新无序区。

这样，n个记录的文件的直接选择排序可经过n-1趟直接选择排序得到有序结果。

直接选择排序是不稳定的。

2．实现

源代码如下。

```
1    #include <iostream>
2    using namespace std;
3
```

```
4    void select_sort(int a[], int len)
5    {
6        int i,j,x,l;
7
8        for(i=0; i<len; i++)              //进行n-1次遍历
9        {
10           x = a[i];                     //每次遍历前x和l的初值设置
11           l = i;
12           for(j=i; j<len; j++)          //遍历从i位置向数组尾部进行
13           {
14               if(a[j] < x)
15               {
16                   x = a[j];     //x保存每次遍历搜索到的最小数
17                   l = j;        //l记录最小数的位置
18               }
19           }
20           a[l] = a[i];                  //把最小元素与a[i]进行交换
21           a[i] = x;
22       }
23   }
24
25   void main()
26   {
27       int data[9] = {54,38,96,23,15,72,60,45,83};
28       select_sort(data, 9);             //快速排序
29       for(int i = 0; i<9; i++)
30           cout << data[i] << " ";       //打印排序后的数组
```

31　}

select_sort 函数进行了 n-1 趟排序。局部变量 x 和 l 分别记录每次遍历时所得的最小元素值及所在位置，代码第20～21行利用它们进行与a[i]的交换。以main函数中的data数组为例，说明其具体步骤。

（1）第1次排序：数组各元素为54，38，96，23，15，72，60，45，83，此时i为0，遍历整个数组得到最小元素15，然后与a[0]进行交换，结果为15，38，96，23，54，72， 60，45，83。

（2）第2次排序：此时i为1，遍历从a[1]开始到数组末尾结束，得到最小元素23，然后与a[1]进行交换，结果为15，23，96，38，54，72，60，45，83。

（3）第3次排序：此时i为2，遍历从a[2]开始到数组末尾结束，得到最小元素38，然后与a[2]进行交换，结果为15，23，38，96，54，72，60，45，83。

显然，每次排序都选出了一个最小的元素，与遍历起始位置的元素进行交换。通过n-1次这样的排序，最终把整个数组进行了排序。

执行结果为：

1　15 23 38 45 54 60 72 83 96

面试题6 编程实现堆排序

考点：堆排序算法的实现

出现频率：★★★

【解析】

1．堆排序

堆排序定义：n个序列Al，A2，…，An称为堆，有下面两种不同类型的堆。

小根堆：所有子结点都大于其父节点，即Ai≤A2i 且Ai≤A2i+1。

大根堆：所有子结点都小于其父节点，即Ai≥A2i 且Ai≥A2i+1。

若将此序列所存储的向量A[1...n]看为一棵完全二叉树的存储结构，则堆实质上是满足如下性质的完全二叉树：树中任一非叶结点的关键字均不大于（或不小于）其左、右子节点（若存在）的关键字。

因此堆排序（HeapSort）是树形选择排序。在排序过程中，将R[l...n]看成一棵完全二叉树的顺序存储结构，利用完全二叉树中双亲结点和孩子结点之间的内在关系，在当前无序区中选择关键字最大（或最小）的记录。

用大根堆排序的基本思想：

（1）先将初始A[1...n]建成一个大根堆，此堆为初始的无序区。

（2）再将关键字最大的记录A[1]（堆顶）和无序区的最后一个记录A[n]交换，由此得到新的无序区A[1...n-1]和有序区A[n]，且满足A[1...n-1]≤A[n]。

（3）由于交换后新的根A[1]可能违反堆性质，故应将当前无序区A[1...n-1]调整为堆。然后再次将A[1...n-1]中关键字最大的记录A[1]和该区间的最后一个记录A[n-1]交换，由此得到新的无序区A[1...n-2]和有序

区A[n-1...n]，且仍满足关系A[1...n-2]≤A[n-1...n]，同样要将A[1...n-2]调整为堆。

（4）对调整的堆重复进行上面的交换，直到无序区只有一个元素为止。

构造初始堆必须用到调整堆的操作，现在说明Heapify函数思想方法。

每趟排序开始前，A[l…i]是以A[1]为根的堆，在A[1]与A[i]交换后，新的无序区A[1…i-1]中只有A[1]的值发生了变化，故除A[1]可能违反堆性质外，其余任何结点为根的子树均是堆。因此，当被调整区间是A[low…high]时，只须调整以A[low]为根的树即可。

可以使用“筛选法”进行堆的调整。A[low]的左、右子树（若存在）均已是堆，这两棵子树的根A[2low]和 A[2low+1]分别是各自子树中关键字最大的节点。若 A[low]不小于这两个孩子节点的关键字，则 A[low]未违反堆性质，以 A[low]为根的树已是堆，无须调整；否则必须将A[low]和它的两个孩子节点中关键字较大者进行交换，即A[low]与A[large]（A[large]=max(A[2low]，A[2low+1])）交换。交换后又可能使节点 A[large]违反堆性质。同样，由于该节点的两棵子树（若存在）仍然是堆，故可重复上述调整过程，对以A[large]为根的树进行调整。此过程直至当前被调整的节点已满足堆性质，或者该节点已是叶子为止。上述过程就像过筛子一样，把较小的关键字逐层筛下去，而将较大的关键字逐层选上来。

2．实现

源代码如下。

```
1    #include <iostream>
2    using namespace std;
3
4    int heapSize = 0;
```

```
5
6  //返回左子节点索引
7  int Left(int index) { return ((index << 1) + 1);}
8
9  //返回右子节点索引
10  int Right(int index) {return ((index << 1) + 2);}
11
12  //交换a、b的值
13  void swap(int *a, int *b) {int temp = *a;*a = *b;*b = temp;}
14
15  //array[index]与其左、右子树进行递归对比
16  //用最大值替换array[index],index表示堆顶索引
17  void maxHeapify(int array[], int index)
18  {
19      int largest = 0;                    //最大数
20      int left = Left(index);             //左子节点索引
21      int right = Right(index);           //右子节点索引
22
23      //把largest赋为堆顶与其左子节点的较大者
24      if ((left <= heapSize) && (array[left] > array[index]))
25          largest = left;
26      else
27          largest = index;
28
29      //把largest与堆顶的右子节点比较，取较大者
30      if ((right <= heapSize) && (array[right] > array[largest]))
31          largest = right;
```

```
32
33      //此时largest为堆顶、左子节点、右子节点中的最大者
34      if (largest != index)
35      {
36          //如果堆顶不是最大者,则交换,并递归调整堆
37          swap(&array[index], &array[largest]);
38          maxHeapify(array, largest);
39      }
40  }
41
42  //初始化堆，将数组中的每一个元素置放到适当的位置
43  //完成之后，堆顶的元素为数组的最大值
44  void buildMaxHeap(int array[], int length)
45  {
46      int i;
47      heapSize = length;              //堆大小赋为数组长度
48      for (i = (length >> 1); i >= 0; i--)
49          maxHeapify(array, i);
50  }
51
52  void heap_sort (int array[], int length)
53  {
54      int i;
55
56      //初始化堆
57      buildMaxHeap(array, (length - 1));
58
```

```
59      for (i = (length - 1); i >= 1; i--)
60      {
61          //堆顶元素array[0]（数组的最大值）被置换到数组的尾部array[i]
62          swap(&array[0], &array[i]);
63          heapSize--;                    //从堆中移除该元素
64          maxHeapify(array, 0);          //重建堆
65      }
66  }
67
68  int main(void)
69  {
70      int a[8] = {45, 68, 20, 39, 88, 97, 46, 59};
71      heap_sort (a, 8);
72      for(int i=0; i<8; i++)
73      {
74          cout << a[i] << " ";
75      }
76      cout << endl;
77      return 0;
78  }
```

heap_sort函数按下面步骤进行。

（1）调用buildMaxHeap对数组进行堆的初始化（代码第57行）。

（2）由于堆顶元素（array[0]）的值是最大的（大根堆），因此把它与数组尾部进行交换。并把heapSize递减1，即从堆中移除数组尾部元素。

（3）由于只有剩下的堆顶元素（array[0]）不满足堆，因此调用

maxHeapify重建堆。

（4）对前面两步进行循环调用，直到堆中只含有堆顶，此时heapSize变为1（i为0）。

其中buildMaxHeap函数初始化堆时也调用了maxHeapify函数，而maxHeapify使用递归的方法把堆调整为大根堆。

测试结果为：

1　20 39 45 46 59 68 88 97

面试题7 实现归并排序的算法（使用自顶向下的方法）

考点：归并排序算法的实现

出现频率：★★★

【解析】

1．归并排序

归并排序（Merge Sort）是利用“归并”技术来进行排序。归并是指将若干个已排序的子文件合并成一个有序的文件。

两路归并算法的基本思路：设两个有序的子文件（相当于输入堆）放在同一向量中相邻的位置上：A[low…m]，a[m+1…high]，先将它们合并到一个局部的暂存向量Temp（相当于输出堆）中，待合并完成后将Temp复制回A[low...high]中。

归并排序有两种实现方法：自底向上和自顶向下。

自底向上方法的基本思想：

（1）第1趟归并排序时，将待排序的文件A[1...n]看作n个长度为1的有序子文件，将这些子文件两两归并。若n为偶数，则得到n/2个长度为2的有序子文件；若n为奇数，则最后一个子文件不参与归并。故本趟归并完成后，前n/2个有序子文件长度为2，但最后一个子文件长度仍为1。

（2）第2趟归并则是将第1趟归并所得到的n/2个有序的子文件两两归并，如此反复，直到最后得到一个长度为n的有序文件为止。

（3）上述每次归并操作，均是将两个有序的子文件合并成一个有序的子文件，故称其为“二路归并排序”。类似地，有k（k>2）路归并排序。

自顶向下算法的设计，形式更为简洁。设归并排序的当前区间是A[low...high]，步骤如下。

（1）分解：将当前区间一分为二，即求分裂点。

（2）求解：递归地对两个子区间A[low...mid]和A[mid+1...high]进行归并排序。

（3）组合：将已排序的两个子区间 A[low...mid]和 A[mid+1...high]归并为一个有序的区间R[low…high]。

（4）递归的终结条件：子区间长度为1（一个记录自然有序）。

2．实现

归并排序算法可用顺序存储结构，也易于在链表上实现。本题中我们使用数组结构。根据前面介绍过的自顶向下算法步骤，可以实现如下程序。

```
1    #include <iostream>
2    using namespace std;
3
4    //将分治的两端按大小次序填入临时数组，最后把临时数组拷贝到原始数组中
5    //lPos到rPos-1为一端,rPos到rEnd为另外一端。
6    void Merge(int a[], int tmp[], int lPos,int rPos, int rEnd )
7    {
8       int i, lEnd, NumElements, tmpPos;
9       lEnd = rPos - 1;
10       tmpPos = lPos;                          //从左端开始
11       NumElements = rEnd - lPos + 1;          //数组长度
12
13       while( lPos <= lEnd && rPos <= rEnd )
14       {
```

15 if(a[lPos] <= a[rPos]) //比较两端的元素值

16 tmp[tmpPos++] = a[lPos++]; //把较小的值先放入tmp临时数组

17 else

18 tmp[tmpPos++] = a[rPos++];

19 }

20

21 //到这里，左端或右端只能有一端还可能含有剩余元素

22 while(lPos <= lEnd) //把左端剩余的元素放入tmp

23 tmp[tmpPos++] = a[lPos++];

24

25 while(rPos <= rEnd) //把右端剩余的元素放入tmp

26 tmp[tmpPos++] = a[rPos++];

27

28 for(i = 0; i < NumElements; i++, rEnd--)

29 a[rEnd] = tmp[rEnd]; //把临时数组拷贝到原始数组

30 }

31

32 void msort(int a[], int tmp[], int low, int high)

33 {

34 if(low >= high) //结束条件，原子结点return

35 return ;

36

37 int middle = (low + high) / 2; //计算分裂点

38 msort(a, tmp, low, middle); //对子区间[low,middle]递归做归并排序

```
39      msort(a, tmp, middle+1, high);          //对子区间[middle+1,high]递归做归并排序
40      Merge(a, tmp, low, middle+1, high);      //组合,把两个有序区合并为一个有序区
41  }
42
43  void merge_sort( int a[], int len )
44  {
45      int *tmp = NULL;
46      tmp = new int[len];                     //分配临时数组空间
47      if(tmp != NULL)
48      {
49          msort(a, tmp, 0, len-1 );           //调用msort归并排序
50          delete []tmp;                       //释放临时数组内存
51      }
52  }
53
54  int main()
55  {
56      int a[8] = {8, 6, 1, 3, 5, 2, 7, 4};
57      merge_sort(a, 8);
58      return 0;
59  }
```

merge_sort 函数是归并的最外层调用，它调用了msort 函数，msort是归并算法的递归实现。它的步骤与前面介绍的相同，分为三个步骤：

（1）代码第37行，计算分裂点，把区间一分为二。

（2）代码第38～39行，递归地对两个子区间A[low...middle]和

A[middle+1...high]进行归并排序。

（3）代码第40行，调用Merge函数合并两个排序后的区间。

Merge函数将分治的两端（这两端是已经排好序的）按大小次序填入临时数组，最后把临时数组拷贝到原始数组中。

面试题8 使用基数排序对整数进行排序

考点：基数排序算法的实现

出现频率：★★

【解析】

1．基数排序

基数排序是箱排序的改进和推广。

箱排序也称桶排序（Bucket Sort），其基本思想是：设置若干个箱子，依次扫描待排序的记录R[0]，R[1]，…，R[n-1]，把关键字等于k的记录全都装入到第k个箱子里（分配），然后按序号依次将各非空的箱子首尾连接起来（收集）。

例如，要将一副混洗的52张扑克牌按点数A<2<…<J<Q<K排序，需设置13个“箱子”，排序时依次将每张牌按点数放入相应的箱子里，然后依次将这些箱子首尾相接，就得到了按点数递增顺序排列的一副牌。

基数排序是基于多关键字的，什么是多关键字呢？如果文件中任何一个记录 R[i]的关键字都由 d 个分量构成，而且这 d 个分量中每个分量都是一个独立的关键字，则文件是多关键字的（比如扑克牌有两个关键字：点数和花色）。

通常实现多关键字排序有两种方法：

最高位优先（Most Significant Digit first，MSD）；

最低位优先（Least Significant Digit first，LSD）。

基数排序是典型的LSD排序方法，其基本思想是：从低位到高位依次对数据进行箱排序。在d趟箱排序中，所需的箱子数就是基数rd（可能的取值个数），这就是“基数排序”名称的由来。

比如，对于值范围为10～99的整数序列：45，13，58，64，29，

74，39，18，使用基数排序需要 10 个箱子（从 0～9 标号）进行分配和收集。我们如果把每一个数看成由两个关键字构成（个位数和十位数），那么可以对它们进行两次分配和收集（分别对于个位和十位），具体步骤如下。

（1）对序列的各个元素按个位进行顺序装箱，即45 装入5 号箱，13装入3 号箱，58和18装入8号箱，64和74装入4号箱，29和39装入9号箱。

（2）从0到9号箱顺序依次收集到原序列，即3号箱的13，4号箱的64和74，5号箱的45，8号箱的58和18，9号箱的29和39被依次收集。序列变为13，64，74，45，58， 18，29，39。

（3）对序列的各个元素按十位进行顺序装箱，即13和18装入1号箱，29装入2号箱， 30装入3号箱，45装入4号箱，58装入5号箱，64装入6号箱，74装入7号箱。

（4）再次从0到9号箱顺序收集到原序列，序列变为13，18，29，30，45，58，64，74。此时完成基数排序。

对于一个两位数来说，其十位数当然比个位数关键。因此使用LSD时，先对个位数开始分配和收集。

2．实现

前面我们已经分析过了使用基数排序对整数序列进行排序的具体步骤。因此，如果整数的范围没有指明，则我们需要查找数组最大的元素有多少位数，以便确定需要进行几次分配和收集，还需要知道每一位是什么。比如数据 167，我们不仅需要知道 167 是一个三位数，而且还需要知道它的个位是7，十位是6，百位是1。

程序代码如下。

```
1    #include <iostream>
2    #include <math.h>
3    using namespace std;
```

```
4
5   int find_max(int a[], int len)              //查找长度为len的数组的最
大元素
6   {
7     int max = a[0];                         //max从a[0]开始
8     for(int i=1; i<len; i++)
9     {
10         if( max < a[i] )                  //如果发现元素比max大，
11               max = a[i];                 //就重新给max赋值
12      }
13      return max;
14   }
15
16   //计算number有多少位
17   int digit_number(int number)
18   {
19      int digit = 0;
20      do
21      {
22            number /= 10;
23            digit++;
24      } while(number != 0);
25      return digit;
26   }
27
28   //返回number上第Kth位的数字
29   int kth_digit(int number, int Kth)
```

```
30  {
31      number /= pow(10, Kth);
32      return number % 10;
33  }
34
35  //对长度为len的数组进行基数排序
36  void radix_sort(int a[], int len)
37  {
38      int *temp[10];                      //指针数组,每一个指针表示一
个箱子
39      int count[10] = {0,0,0,0,0,0,0,0,0,0};   //用于存储每个箱子装
有多少元素
40      int max = find_max(a, len);             //取得序列中的最大整数
41      int maxDigit = digit_number(max);       //得到最大整数的位数
42      int i, j, k;
43      for(i=0; i<10; i++)
44      {
45          temp[i] = new int[len];             //使每一个箱子能装下len个
int元素
46          memset(temp[i], 0, sizeof(int) * len);   //初始化为0
47      }
48      for(i=0; i<maxDigit; i++)
49      {
50          memset(count, 0, sizeof(int) * 10); //每次装箱前把count清空
51          for(j=0; j<len; j++)
52          {
53              int xx = kth_digit(a[j], i);   //将数据安装位数放入到暂
```

存数组中

```
54                  temp[xx][count[xx]] = a[j];
55                  count[xx]++;              //此箱子的计数递增
56              }
57              int index = 0;
58              for(j=0; j<10; j++)           //将数据从暂存数组中取回，放
```
入原始数组中
```
59              {
60                  for(k=0; k<count[j]; k++)   //把箱子里所有的元素都取
```
回到原始数组
```
61                  {
62                      a[index++] = temp[j][k];
63                  }
64                }
65          }
66      }
67
68      int main(void)
69      {
70          int a[] = {22, 32, 19, 53, 47, 29};
71          radix_sort(a, 6);
72
73          return 0;
74      }
```

下面简单说明一下radix_sort函数的执行步骤。

（1）代码第 40～41 行，调用 find_max 取得序列中的最大整数，并调用 digit_number得到其最大位数maxDigit。

（2）代码第43～47行，分配10个足够大的箱子来存放序列中的整数。

（3）代码第51～56行，针对序列中整数的个位数，进行第一次分配箱子。

（4）代码第57～64行，依次收集每个箱子的元素，放回到原始数组中。

步骤（3）和步骤（4）一共需要进行maxDigit次，每一次针对序列中整数的不同位数进行分配箱子。代码第53行调用了kth_digit计算元素各个位数的数字，以确定放入哪一个箱子。另外，在 radix_sort 函数里还有一个局部数组 count，它被用来在每次分配箱子后，保存各箱子里所含整数元素的个数。

面试题9 选择题——各排序算法速度的性能比较

考点：各排序算法速度的性能比较

出现频率：★★★★

下面哪种排序法对1，2，3，5，4最快？（　　）

A．quick sort

B．buble sort

C．merge sort

【解析】

选择排序算法的时候，需要考虑以下几点。

数据的规模；

数据的类型；

数据已有的顺序。

一般来说，当数据规模较小时，应选择直接插入排序或冒泡排序。任何排序算法在数据量小时基本体现不出差距。

考虑数据的类型，比如全部是正整数时，应该考虑使用桶排序。

考虑数据已有顺序，快速排序是一种不稳定的排序（当然可以改进）。对于大部分排好的数据，快速排序会浪费大量不必要的步骤。我们说快速排序好，是指大量随机数据下，使用快速排序的效果最理想，而不是指所有情况。

根据题目来看，1，2，3，5，4数据量极小，已经基本排好序。所以，此时冒泡排序是最佳选择。

【答案】

B

面试题10 各排序算法的时间复杂度的比较

考点：各排序算法的时间复杂度的比较

出现频率：★★★

写出下列算法的时间复杂度。

（1）冒泡排序；

（2）选择排序；

（3）插入排序；

（4）快速排序；

（5）堆排序；

（6）归并排序。

【答案】

冒泡排序算法的时间复杂度是O(n^2)。

选择排序算法的时间复杂度是O(n^2)。

插入排序算法的时间复杂度是O(n^2)。

快速排序是不稳定的，最理想情况下的算法时间复杂度是O(nlog2n)，最坏是O(n^2)。

堆排序算法的时间复杂度是O(nlogn)。

归并排序算法的时间复杂度是O(nlog2n)。

第10章 泛型编程

大家都知道C++是C的超集，具有面向对象编程的能力。然而许多程序员可能并不知道，C++不仅是一个面向对象程序语言，它还适用于泛型编程（Generic Programming）。这项技术可以大大增强你的能力，协助你写出高效率并可重复运用的软件组件（Software Components）。

泛型编程是一种新的编程思想，它基于模板技术，有效地将算法与数据结构分离，降低了模块间的耦合度。

面试题1 举例说明什么是泛型编程

考点：泛型编程的基本概念

出现频率：★★★

【解析】

泛型编程指编写完全一般化并可重复使用的算法，其效率与针对某特定数据类型而设计的算法相同。所谓泛型，是指具有在多种数据类型上皆可操作的含意，在 C++中实际上就是使用模板实现。

举一个简单的例子，比如我们要比较两个数的大小，这两个数的类型可能是 int，也可能是float，还有可能是double。一般编程时我们可能这样写函数（不考虑使用宏的情况）：

```
1   int max(int a, int b)              //参数和返回值都是int类型
2   {
3       return a > b? a: b;
4   }
5
6   float max(float a, float b)          //参数和返回值都是float类型
7   {
8       return a > b? a: b;
9   }
10
11   double max(double a, double b)       //参数和返回值都是double类型
12   {
13       return a > b? a: b;
```

```
14   }
```

可以看到，上面写了 3 个重载函数，它们的区别仅仅是类型（参数及返回值）不同，而函数体完全一样。

而如果使用模板，我们可以这样写：

```
1   template <class T>          //class也可用typename替换
2   T max(T a, T b)          //参数和返回值都是T类型
3   {
4       return a > b? a: b;
5   }
```

这里的class不代表对象的类，而是类型（可用typename替换）。这样max函数的各个参数以及返回值类型都为 T。对于任意类型的两个数，我们都可以调用 max 求大小，测试代码如下。

```
1   int main()
2   {
3       cout << max(1, 2) << endl;          //隐式调用int类型的max
4       cout << max(1.1f, 2.2f) << endl;    //隐式调用float类型的max
5       cout << max(1.11l, 2.22l) << endl;    //隐式调用double类型的max
6       cout << max('A', 'C') << endl;       //隐式调用char类型的max
7       cout << max<int>(1, 2.0) << endl;    //必须指定int类型
8
9       return 0;
10  }
```

程序执行结果如下。

```
1   2
2   2.2
3   2.22
```

4　C

5　2

上面的程序中对于int、float、double以及char类型都进行了测试。这里需要注意的是，第7行的测试中显示地指定了类型为int，这是因为参数1为int类型，参数2.0为double类型，此时如果不指定使用什么类型，会产生编译的模糊性，即编译器不知道是需要调用 int版本的max还是调用double版本的max函数。

另外，T 作为 max 函数的各参数以及返回值类型，它几乎可以是任意类型，即除了基本类型（int、float、char等）外，还可以是类。当然，这个类需要重载“>”操作符（因为max函数体使用了这个操作符）。

显然，使用泛型编程（模板）可以极大地增加代码的重用性。

面试题2 函数模板与类模板分别是什么

考点：函数模板与类模板的基本概念和区别

出现频率：★★★

【解析】

1．什么是函数模板和类模板？

函数模板是一种抽象的函数定义，它代表一类同构函数。通过用户提供的具体参数， C++编译器在编译时刻能够将函数模板实例化，根据同一个模板创建出不同的具体函数。这些函数之间的不同之处主要在于函数内部一些数据类型的不同，而由模板创建的函数的使用方法与一般函数的使用方法相同。函数模板的定义格式如下。

```
1    template<TYPE_LIST,ARG_LIST>Function_Definition
```

其中，Function_Definition为函数定义；TYPE_LIST被称为类型参数表，是由一系列代表类型的标识符组成的，其间用逗号分隔，这些标识符的风格通常是由大写字母组成；ARG_LIST 称为变量表，其中含有由逗号分隔开的多个变量说明，相当于一般函数定义中的形式参数。前面面试题中的max就是函数模板的一个例子，因此这里不再另外举例。

C++提供的类模板是一种更高层次的抽象的类定义，用于使用相同代码创建不同的类模板的定义与函数模板的定义类似，只是把函数模板中的函数定义部分换作类说明，并对类的成员函数进行定义即可。在类说明中可以使用出现在TYPE_LIST中的各个类型标识以及出现在ARG_LIST中的各变量。

```
1    template<<棋板参数表>>
2    class<类模板名>
3    {<类模板定义体>},
```

例如，我们需要定义一个表示平面的点（Point）类，这个类有两个成员变量，分别表示横坐标和纵坐标，并且这两个坐标的类型可以是int、float、double 等类型。因此可能写出类似Point_int_int、Point_float_int、Point_float_float等这样的类。通过类模板，我们只需要写一个类。

```
1   #include <iostream>
2   using namespace std;
3
4   template <class T1, class T2>
5   class Point_T
6   {
7   public:
8      T1 a;              //成员a为T1类型
9      T2 b;              //成员b为T2类型
10       Point_T() : a(0), b(0) {}    //默认构造函数
11       Point_T(T1 ta, T2 tb) : a(ta), b(tb) {}          //带参数的构造函数
12       Point_T<T1, T2>& operator=(Point_T<T1, T2> &pt);    //赋值函数
13       friend Point_T<T1,T2> operator +(Point_T<T1,T2> &pt1, Point_T<T1, T2> &pt2); //重载+
14   };
15
16   template <class T1, class T2>
17   Point_T<T1, T2>& Point_T<T1, T2>::operator=(Point_T<T1, T2> &pt)       //赋值函数
18   {
```

```
19        this->a = pt.a;
20        this->b = pt.b;
21        return *this;
22    }
23
24    template <class T1, class T2>
25    Point_T<T1, T2> operator +(Point_T<T1, T2> &pt1, Point_T<T1, 
T2> &pt2)//重载+
26    {
27        Point_T<T1, T2> temp;
28        temp.a = pt1.a + pt2.a; //结果对象中的a和b分别为两个参数对
象的各自a和b之和
29        temp.b = pt1.b + pt2.b;
30        return temp;
31    }
32
33    template <class T1, class T2>
34    ostream& operator << (ostream& out, Point_T<T1, T2>& pt)    //
重载输出流操作符
35    {
36        out << "(" << pt.a << ", ";                    //输出(a, b)
37        out << pt.b << ")";
38        return out;
39    }
40
41    int main()
42    {
```

```
43      Point_T<int, int> intPt1(1, 2);                 //T1和T2都是int
44      Point_T<int, int> intPt2(3, 4);                 //T1和T2都是int
45      Point_T<float, float> floatPt1(1.1f, 2.2f);     //T1和T2都是float
46      Point_T<float, float> floatPt2(3.3f, 4.4f);     //T1和T2都是float
47
48      Point_T<int, int> intTotalPt;
49      Point_T<float, float> floatTotalPt;
50
51      intTotalPt = intPt1 + intPt2;       //类型为Point_T<int, int>的对象相加
52      floatTotalPt = floatPt1 + floatPt2; //类型为Point_T<float, float>的对象相加
53
54      cout << intTotalPt << endl;       //输出Point_T<int, int>的对象
55      cout << floatTotalPt << endl;       //输出Point_T<float, float>的对象
56
57      return 0;
58  }
```

Point_T类就是一个类模板，它的成员a和b分别为T1和T2类型。这里我们还实现了它的构造函数、赋值函数、“+”运算符的重载以及输出流操作符“<<”的重载。

使用Point_T类非常方便，我们可以进行各种类型的组合。

代码第43、44 行，定义了两个Point_T<int, int>类的对象intPt1 和intPt2，表明这两个对象的成员a和b都是int类型。

代码第45、46 行，定义了两个Point_T<float, float>类的对象floatPt1和floatPt2，表明这两个对象的成员a和b都是float类型。

代码第51行，对intPt1和intPt2进行对象加法，结果保存到intTotalPt中。此过程先调用“+”函数，再调用“=”函数。

代码第 52 行，与第 51 行类似，只是相加的对象和结果对象都是Point_T<float, float>类的对象。

代码第54、55行，输出对象intTotalPt和floatTotalPt的内容。

可以看出，通过使用类模板Point_T可以创建不同的类，大大提高了代码的可维护性以及可重用性。

有一些概念需要区别：函数模板与模板函数、类模板和模板类是不同的意思。

函数模板的重点是模板，它表示的是一个模板，用来生产函数。例如前面面试题的max是一个函数模板。而模板函数的重点是函数，它表示的是由一个模板生成的函数。例如max<int>，max<float>等都是模板函数。

类模板和模板类的区别与上面的类似。类模板用于生产类，例如Point_T就是一个类模板。而模板类是由一个模板生成的类，例如Point_T<int, int>和 Point_T<float_float>都是模板类。

2．函数模板和类模板有什么区别？

在面试题1的程序代码中，我们在使用函数模板max时不一定必须指明T的类型，函数模板max 的实例化是由编译程序在处理函数调用时自动完成的。当调用max(1, 2)时自动生成实例max<int>，而调用max(1.1f, 2.2f)时自动生成实例max<float,float>。当然，也可以显示指定T的类型。

对于本面试题的类模板 Point_T 来说，其实例化必须被显示地指定，比如 Point_T<int, int>、Point_T<float, float >。

【答案】

函数模板是一种抽象的函数定义，它代表一类同构函数。类模板是一种更高层次的抽象的类定义。

函数模板的实例化是由编译程序在处理函数调用时自动完成的，而类模板的实例化必须由程序员在程序中显式地指定。

面试题3 使用模板有什么缺点？如何避免

考点：理解模板编程的缺陷

出现频率：★★★

【解析】

templates（模板）是节省时间和避免代码重复的极好的方法。我们可以只输入一个类模板，就能让编译器实例化所需要的很多个特定类及函数。类模板的成员函数只有被使用时才会被实例化，所以只有在每一个函数都在实际中被使用时，我们才会得到这些函数。

这确实是一个很重要的技术，但是如果不小心，使用模板可能会导致代码膨胀。什么是代码膨胀？请看下面的例子。

```
1    template <class T, int num>
2    class A
3    {
4    public:
5        void work()
6        {
7                cout << "work() " << endl;
8                cout << num << endl;
9        }
10   };
11
12   int main()
13   {
14       A<int, 1>v1;
```

```
15      A<int, 2>v2;
16      A<int, 3>v3;
17      A<int, 4>v4;
18      v1.work();
19      v2.work();
20      v3.work();
21      v4.work();
22      return 0;
23   }
```

类模板 A 取得一个类型参数 T，并且它还有一个类型为 int 的参数、一个非类型参数（non-type parameter）。与类型参数相比，虽然非类型参数不是很通用，但它们是完全合法的。在本例中，由于num的不同，代码第14行到第17行的调用生成了三个A的实例，然后在第18～21行又生成了不同的函数调用。

虽然这些函数做了相同的事情（打印一个“work()”和 num），但它们却有不同的二进制代码。这就是所说的由于模板导致的代码膨胀。也就是说，虽然源代码看上去紧凑而整洁，但是目标代码却臃肿而松散，会严重影响程序的运行效率。

如何避免这种代码膨胀呢？有一个原则，就是把 C++模板中与参数无关的代码分离出来。也就是让与参数无关的代码只有一份复制。对类模板A可以进行如下修改。

```
1   template <class T>
2   class Base
3   {
4   public:
5     void work(int num)
6     {
```

```
7               cout << "work ";
8               cout << num << endl;
9         }
10    };
11
12    template<class T, int num>
13    class Derived : public Base<T>
14    {
15    public:
16        void work()
17        {
18            Base<T>::work(num);
19        }
20    };
21
22    int main()
23    {
24        Derived<int, 1>d1;
25        Derived<int, 2>d2;
26        Derived<int, 3>d3;
27
28        d1.work();
29        d2.work();
30        d3.work();
31        return 0;
32    }
```

程序中work的参数版本是在一个Base类（基类）中的。与Derived

类一样，Base类也是一个类模板。但是与Derived类不一样的是，它参数化的仅仅是类型T，而没有num。因此，所有持有一个给定类型的Derived将共享一个单一的Base类。比如代码第24～26行实例化的模板类都共享 Base<int>模板类，从而它们的成员函数都共享 Base<int>模板类中的work这个单一的复制。

【答案】

模板的缺点：不当地使用模板会导致代码膨胀，即二进制代码臃肿而松散，会严重影响程序的运行效率。

解决方法：把C++模板中与参数无关的代码分离出来。

面试题4 选择题——类模板的实例化

考点：对类模板的实例化的理解

出现频率：★★★★

```
1    template<class T, int size = 10>
2    class Array {
3    …
4    };
5    void foo( )
6    {
7        Array <int> arr1;
8        Array <char> arr4, arr5;
9        Array <int> arr2, arr3;
10       Array <double> arr6;
11   …
12   }
```

How many instances of the template class Array will get instantiated inside the function foo?（　　）

A．3 B．6 C．4 D．1

【解析】

模板类（template class）的实例个数是由类型参数的种类决定的。代码第7 行和第9 行实例化的模板类都是Array<int, 10>，代码第8 行实例化的模板类是Array<char, 10>，代码第10 行实例化的模板类是Array<double, 10>。一共是3 个实例。

【答案】

A

面试题5 解释什么是模板的特化

考点：对模板的特化的理解

出现频率：★★★

【解析】

模板的特化（template specialization）分为两类：函数模板的特化和类模板的特化。

（1）函数模板的特化：当函数模板需要对某些类型进行特别处理时，称为函数模板的特化。例如

```
1    bool IsEqual(T t1, T t2)
2    {
3        return t1 == t2;
4    }
5
6    int main()
7    {
8        char str1[] = "Hello";
9        char str2[] = "Hello";
10       cout << IsEqual(1, 1) << endl;
11       cout << IsEqual(str1, str2) << endl;    //输出0
12       return 0;
13   }
```

代码第11 行比较字符串是否相等。由于对于传入的参数是char *类型的，max 函数模板只是简单地比较了传入参数的值，即两个指针是否相等，因此这里打印0。显然，这与我们的初衷不符。因此，max 函数

模板需要对char *类型进行特别处理，即特化：

```
1    template <>
2    bool IsEqual(char* t1, char* t2)        //函数模板特化
3    {
4        return strcmp(t1, t2) == 0;
5    }
```

这样，当 IsEqual 函数的参数类型为 char* 时，就会调用 IsEqual 特化的版本，而不会再由函数模板实例化。

（2）类模板的特化：与函数模板类似，当类模板内需要对某些类型进行特别处理时，使用类模板的特化。例如

```
1    template <class T>
2    class compare
3    {
4    public:
5        bool IsEqual(T t1, T t2)
6        {
7                return t1 == t2;
8        }
9    };
10
11    int main()
12    {
13        char str1[] = "Hello";
14        char str2[] = "Hello";
15        compare<int> c1;
16        compare<char *> c2;
17        cout << c1.IsEqual(1, 1) << endl;        //比较两个int类型的参数
```

```
18      cout << c2.IsEqual(str1, str2) << endl;   //比较两个char *类型的参数
19      return 0;
20  }
```

这里代码第18行也是调用模板类compare<char*>的IsEqual进行两个字符串比较。显然这里存在的问题和上面函数模板中的一样，我们需要比较两个字符串的内容，而不是仅仅比较两个字符指针。因此，需要使用类模板的特化：

```
1  template<>
2  class compare<char *> //特化(char*)
3  {
4  public:
5     bool IsEqual(char* t1, char* t2)
6     {
7          return strcmp(t1, t2) == 0;      //使用strcmp比较字符串
8     }
9  };
```

注意：进行类模板的特化时，需要特化所有的成员变量及成员函数。

面试题6 部分模板特例化和全部模板特例化有什么区别

考点：对部分模板特例化和全部模板特例化的区别的理解

出现频率：★★★

【解析】

模板有两种特例化：部分模板特例化和全部模板特例化。

全部模板特例化就是模板中的模板参数全被指定为确定的类型，也就是定义了一个全新的类型。全部模板特例化的类中的函数可以与模板类不一样，例如

```
1    template <class A, class B, class C>
2    class X {};    //(a)
3    template <>
4    class X<int, float, string> {};    //(b)
```

若编译器遇到X<int, float, string>的模板实例化请求，则使用特例化后的版本，即（b）。其他任何类型的组合都是用基本模板，即（a）。

部分模板特例化就是模板中的模板参数没有被全部确定，需要编译器在编译时进行确定。它通常有两种基本情况：

（1）对部分模板参数进行特例化：

```
1    template < class B, class C>
2    class X <int, B, C> {...};    //    (c)
```

当编译器遇到X <int, float, string> 的模板实例化请求时，使用这个特例化的版本(c)。而当编译器遇到X <int, double, char> 时，也是用这个特例化版本。也就是说，只要X<>实例化时，第一个模板实参是int，就使用特例化版本。

（2）使用具有某一特征的类型，对模板参数进行特例化：

```
1    template <class T>
2    class Y {...};        //(d)
3    template <class T>
4    class Y<T*> {...};    //(e)
```

当编译器遇到Y<int*>时，使用特例化模板(e)。当编译器遇到T<float*>时，也使用特例化模板（e）。而其他类型的实例化，如Y<int>，则使用基本模板（d）。也就是说，当模板实参符合特例化版本所需的特征时（在上面例子中是某个类型的指针），则使用特例化版本。

这两种情况有时会混合使用，比如

```
1    template <class A, class B>
2    class Z {...};    //(f)
3    template <typename A>
4    class Z <A&, char> {...};    //(g)
```

当编译器遇到Z<int&, char>或者Z<string&, char>时，使用特例化模板（g）。其他情况使用基本模板（f），比如Z<int&, float>或Z<int, char>等。

面试题7 使用函数模板对普通函数进行泛型化

考点：函数模板的使用

出现频率：★★★

Below is usual way we find one element in an array.（下面是在一个数组中查找一个元素的方法。）

```
1   const int *find1(const int* array, int n, int x)
2   {
3       const int* p = array;
4       for(int i = 0; i < n; i++)
5       {
6               if(*p == x)
7               {
8                   return p;
9               }
10             ++p;
11         }
12         return 0;
13   }
```

In this case we have to bear the knowledge of value type "int", the size of array, even the existence of an array. Would you re-write it using template to eliminate all these dependencies?（这里我们的数组大小，甚至数组的类型都必须是 int 类型的。请你使用模板重写它来消除这些限制。）

【解析】

根据题意，我们需要把find1函数泛型化。显然，find1函数的作用是查找长度为n的array数组中值为x的元素。因此这里只需要把数组类型以及数组各个元素类型泛型化。代码如下。

```
1   template <class T>
2   const T *find1(const T* array, int n, T x)   //返回值,array，x的类型都变成T
3   {
4       const T* p = array;                    //指针类型变成T*
5       for(int i = 0; i < n; i++)
6       {
7           if(*p == x)
8           {
9               return p;
10          }
11              ++p;
12       }
13       return 0;
14   }
```

经过泛型化之后，可以使用find1函数对各种类型的数组进行查找。比如下面的程序分别对int和double类型的数组进行了测试。

```
1   int main()
2   {
3       int intArr[] = {1, 2, 3, 4, 5, 6, 7};
4       double doubleArr[] = {1.1,2.2,3.3,4.4,5.5,6.6};
5       const int *p = find1(intArr, 6, 6);         //对整型数组进行查找
6       cout << *p << endl;
7       const double *q = find1(doubleArr, 6, 6.6);   //对浮点型数组进
```

行查找

```
8       cout << *q << endl;
9       return 0;
10    }
```

测试结果：

```
1    6
2    6.6
```

面试题8 使用类模板对类进行泛型化

考点：类模板的使用

出现频率：★★★

Give an example of implementing a Stack in the template way(only template class declaration without detail definition and realization).

【解析】

原题的翻译：给出一个使用模板实现一个Stack的例子（只需要类声明，而不需要具体的定义和实现）。

Stack（栈）的实现可以用数组，也可以用链表。这里我们用动态数组的方法定义栈。代码如下。

```
1    template <class T>
2    class Stack
3    {
4    public:
5        Stack(int len = 10) ;                   //构造函数，默认数组大小为10
6    ～Stack() { delete [] stackPtr ; }        //析构函数，释放数组内存
7        int push(const T&);    //进栈
8        int pop(T&);           //出栈
9        int isEmpty()const { return top == -1 ; }        //判断栈空
10       int isFull() const { return top == size - 1 ; }    //判断栈满
11   private:
12       int size ;           //栈中的元素个数
13       int top ;           //栈顶元素位置
14       T* stackPtr ;           //保存动态数组指针,类型为T*
```

```
15    };
```

由于使用了模板，进栈（push）函数的参数、退栈（pop）函数的参数以及用于保存动态数组指针的stackPtr的类型都必须是T。这样，我们实现的Stack可以用于各种类型的元素，如int、float、类等。

面试题9 通过类模板设计符合要求的公共类

设计一个公共的class，通过它的接口可以对任何类型的数组排序。

考点：类模板的使用

出现频率：★★★

【解析】

解决本题的关键是对任何类型的数组进行排序。对于不同类型的数组排序，以往我们可以通过不同类型参数的重载函数来完成，例如在本题中，我们可以在这个公共的class中书写数组类型为 int、float、double 等的重载函数。显然，这种做法会导致程序代码重复较多，并且不易于维护，尤其当需要对类对象进行排序的时候。

因此，本题需要使用模板技术来实现排序运算。为了方便，本题中使用了冒泡排序法。

```
1    template <class T>
2    class Test
3    {
4    public:
5        static void Sort(T *array, int len, bool (*Compare)(T& a, T& b))
6        {
7                T temp;                              //用于冒泡排序的交换
8                assert(len >= 1);                    //len必须大于1
9                for (int i=0; i<len-1; i++)          //冒泡排序
10                 {
11                       for (int j=len-1; j>i; j--)
12                       {
```

```
13                    //使用Compare函数指针的方式进行比较
14                    if (Compare(array[j], array[j-1]))
15                    {
16                        temp = array[j-1];   //根据升序或降序需要进行交换
17                        array[j-1] = array[j];
18                        array[j] = temp;
19                    }
20                 }
21            }
22       }
23   };
```

对上面Test类中的Sort成员函数有两点需要说明。

Sort 成员是static 的，因此直接用Test<T>::Sort 访问。

Sort的第3个参数Compare是一个函数指针，它指向下面两个函数模板中的任意一个。

```
1   template <class T>
2   bool ascend(T& a, T& b) //升序，a比b小时返回true
3   {
4      return a < b? true: false;
5   };
6
7   template <class T>
8   bool descend(T& a, T& b) //降序，a比b大时返回true
9   {
10       return a > b? true: false;
11   };
```

函数模板ascend和descend的参数a和b分别代表数组中的两个元素。当Compare指向ascend时，表示按升序排序；当Compare指向descend时，表示按降序排序。

我们可以用Test类的Sort函数对任意类型的数组进行排序，比如int、float、double、类对象的数组等。注意，如果需要对某个类对象的数组排序，我们需要重载这个类的“<”和“>”操作符。例如下面的MyRect定义。

```
1    template <class T>
2    class MyRect
3    {
4    public:
5        MyRect():length(0), width(0) {}
6        MyRect(T len, T wid):length(len), width(wid) {}
7        T Area() { return length * width; }           //返回面积
8    private:
9        T length;    //矩形的长
10       T width;    //矩形的宽
11   };
12
13   template<class T>
14   operator > (MyRect<T>& rect1, MyRect<T>& rect2)    //重载>运算符
15   {
16       return rect1.Area() > rect2.Area()? true: false;
17   }
18
19   template<class T>
```

```
20  operator < (MyRect<T>& rect1, MyRect<T>& rect2)   //重载<运算符
21  {
22      return rect1.Area() < rect2.Area()? true: false;
23  }
```

MyRect（矩形）是一个类模板，它含有两个私有成员，分别为length（长）和width（宽）。并且为了比较两个 MyRect 模板类的对象，我们重载了“<”和“>”操作符。因此，对于MyRect类模板实例化的模板类的对象数组，就能使用前面的Test的Sort成员函数进行排序。

测试代码如下。

```
1  int main()
2  {
3      int int_array[10] = {4, 3, 7, 6, 2, 1,            //int数组定义
4                           9, 8, 5, 10};
5      float float_array[10] = {4.0f, 3.0f, 7.0f, 6.0f,     //float数组定义
6                           2.0f, 1.0f, 9.0f, 8.0f, 5.0f, 10.0f};
7      MyRect<int> rect_array[4] = { MyRect<int>(3, 4),     //MyRect对象数组定义
8                                    MyRect<int>(5, 6),
9                                    MyRect<int>(4, 6),
10                                  MyRect<int>(3, 5) };
11
12      Test<int>::Sort(int_array, 10, descend<int>);         //int数组降序
13      Test<int>::Sort(int_array, 10, ascend<int>);          //int数组升序
14
15      Test<float>::Sort(float_array, 10, descend<float>) ;   //float数组
```

```
降序
16      Test<float>::Sort(float_array, 10, ascend<float>);    //float数组
升序
17
18      Test< MyRect<int> >::Sort(rect_array, 4,
descend<int>);//MyRect数组降序
19      Test< MyRect<int> >::Sort(rect_array, 4,
ascend<int>);        //MyRect数组升序
20
21      return 0;
22   }
```

我们对int、float、MyRect数组分别进行了升序和降序的操作。这里就不再输出各个数组经过每次排序后的结果了，有兴趣的读者可以自己调试查看。

第11章 STL（标准模板库）

STL（Standard Template Library），即标准模板库，它涵盖了常用的数据结构和算法，并且具有跨平台的特点。将泛型编程思想和STL库用于系统设计中，明显降低了开发强度，提高了程序的可维护性及代码的可重用性。这也是越来越多的笔试和面试中考查STL相关知识的原因。

STL是C++标准函数库的一部分。STL的基本观念就是把数据和操作分离。图11.1所示的是STL组件之间的合作。

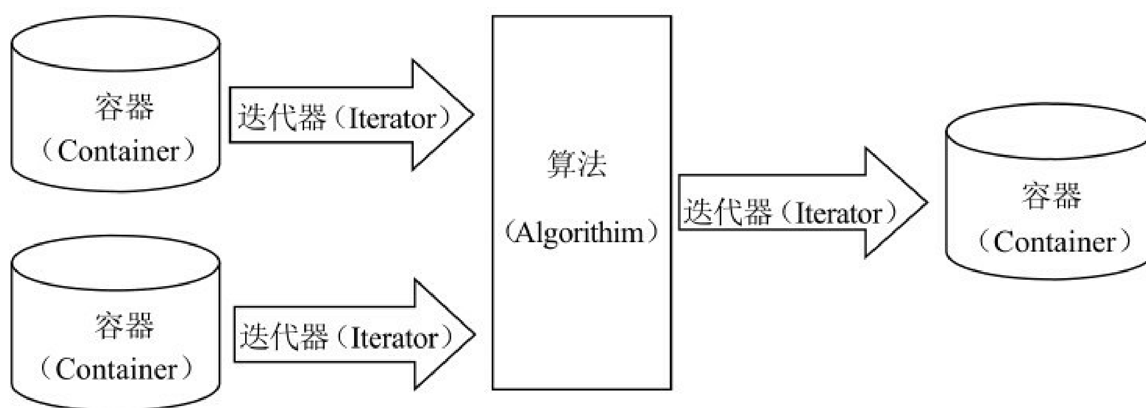


图11.1 STL组件之间的合作

由图11.1可知，STL中数据由容器类别来加以管理，操作则由可定制的算法来完成。迭代器在容器和算法之间充当黏合剂，它使得任何算法都可以和任何容器进行交互运作。STL含有容器、算法、迭代器组件。

其中容器又分为下面几种。

STL 序列容器：vector、string、deque 和list。

STL 关联容器：set、multiset、map 和multimap。

STL 适配容器：stack、queue 和priority_queue。

面试题1 什么是STL

考点：对C++标准模板库的理解

出现频率：★★★★

【解析】

STL（Standard Template Library），即标准模板库，是一个具有工业强度的、高效的C++程序库。它被容纳于C++标准程序库（C++ Standard Library）中，是ANSI/ISO C++标准中最新的也是极具革命性的一部分。该库包含了诸多在计算机科学领域里所常用的基本数据结构和基本算法，为广大C++程序员提供了一个可扩展的应用框架，高度体现了软件的可复用性。它类似于 Microsoft Visual C++中的MFC（Microsoft Foundation ClassLibrary）。

从逻辑层次来看，在STL 中体现了泛型化程序设计的思想（Generic Programming），引入了许多新的名词，比如需求（requirements）、概念（concept）、模型（model）、容器（container）、算法（algorithmn）、迭代器（iterator）等。

从实现层次看，整个STL 是以一种类型参数化（type parameterized）的方式实现的，这种方式基于一个在早先 C++标准中没有出现的语言特性——模板（template）。如果查阅任何一个版本的STL源代码，你就会发现，模板作为构成整个STL的基石是一件千真万确的事情。除此之外，还有许多C++的新特性为STL的实现提供了方便。

STL是最新的C++标准函数库中的一个子集，它占据了整个库的大约80%的分量。而作为在实现 STL 过程中扮演关键角色的模板，则充斥了几乎整个 C++标准函数库。C++标准函数库中的各个组件的关系图如

图11.2所示。

C++标准函数库包含了如下几个组件。

（1）C 标准函数库，尽管有了一些变化，但是基本保持了与原有C 语言程序库的良好兼容。C++标准库中存在两套C的函数库，一套是带有.h扩展名的（比如<stdio.h>），而另一套则没有（比如<cstdio>）。它们确实没有太大的不同。

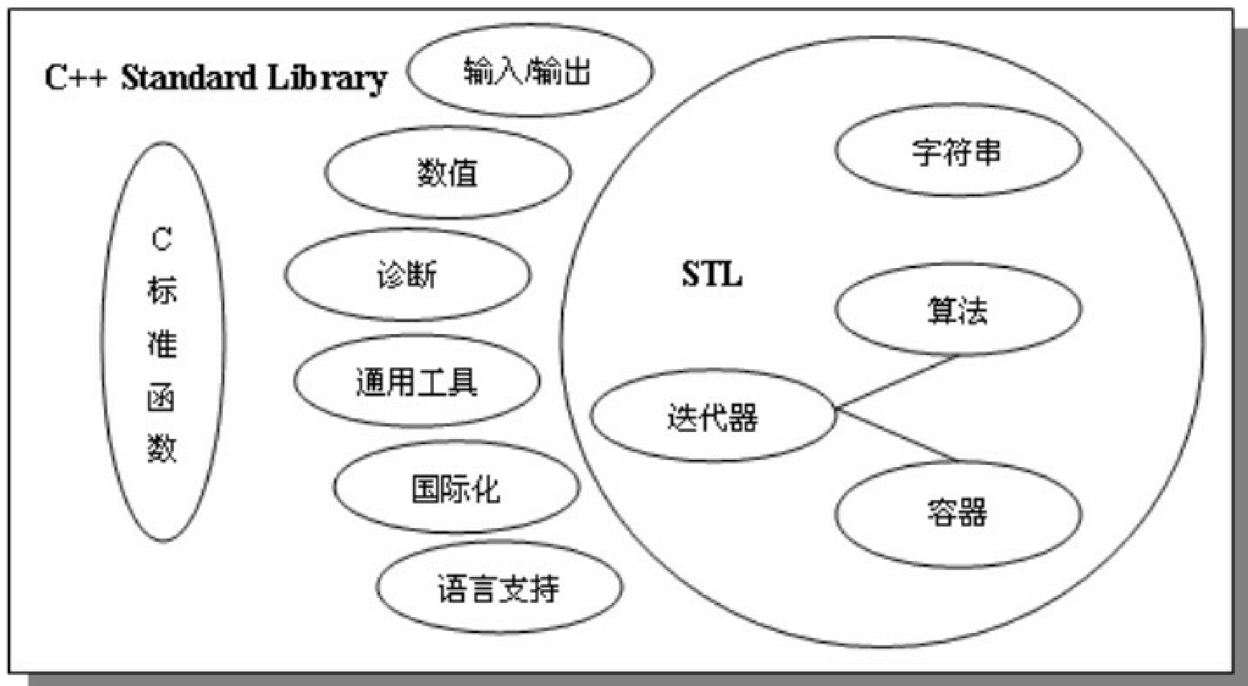


图11.2 C++标准库中各组件关系

（2）语言支持（language support）部分，包含了一些标准类型的定义以及其他特性的定义。这些内容被用于标准库的其他地方或是具体的应用程序中。

（3）诊断（diagnostics）部分，提供了用于程序诊断和报错的功能，包含了异常处理（exception handling）、断言（assertions）、错误代码（error number codes）三种方式。

（4）通用工具（general utilities）部分，这部分内容为C++标准库的其他部分提供支持。当然，你也可以在自己的程序中调用相应功能，比如动态内存管理工具、日期/时间处理工具。记住，这里的内容也已

经被泛化了（采用了模板机制）。

（5）字符串（string）部分，用来代表和处理文本。它提供了足够丰富的功能。

（6）国际化（internationalization）部分，作为OOP特性之一的封装机制在这里扮演着消除文化和地域差异的角色，采用locale和facet可以为程序提供众多国际化支持，包括对各种字符集的支持、日期和时间的表示、数值和货币的处理等等。

（7）容器（containers）部分，是STL的一个重要组成部分，涵盖了许多数据结构，如链表、vector（类似于大小可动态增加的数组）、queue（队列）、stack（堆栈）……string也可以看作一个容器，适用于容器的方法同样也适用于string。

（8）算法（algorithms）部分，是STL的一个重要组成部分，包含了大约70个通用算法，用于操控各种容器，同时也可以操控内建数组。

（9）迭代器（iterators）部分，是STL 的一个重要组成部分。它使得容器和算法能够完美地结合。事实上，每个容器都有自己的迭代器，只有容器自己才知道如何访问自己的元素。它有点像指针，算法通过迭代器来定位和操控容器中的元素。

（10）数值（numerics）部分，包含了一些数学运算功能，提供了复数运算的支持。

（11）输入/输出（input/output）部分，就是经过模板化了的原有标准库中的iostream部分。它提供了对C++程序输入/输出的基本支持。在功能上保持了与原有iostream的兼容，增加了异常处理的机制，并支持国际化（internationalization）。

总体上，在C++标准函数库中，STL主要包含了容器、算法、迭代器。string也可以算作STL的一部分。

【答案】

STL，即标准模板库，是一个具有工业强度的、高效的 C++程序

库。它是最新的 C++标准函数库中的一个子集，包括容器、算法、迭代器组件。

面试题2 具体说明STL如何实现vector

考点：对vector的理解及其实现细节

出现频率：★★★★

【解析】

vector的定义如下。

```
1    template<class _Ty, class _A = allocator<_Ty> >
2    class vector {
3    ……
4    };
```

这里省略了中间的成员。其中_Ty类型用于表示vector中存储的元素类型，_A默认为allocator<_Ty>类型。

这里需要说明的是allocator类，它是一种“内存配置器”，负责提供内存管理（可能包含内存分配、释放、自动回收等能力）相关的服务。于是对于程序员来说，就不用关心内存管理方面的问题了。

vector支持随机访问，因此为了效率方面的考虑，它内部使用动态数组的方式实现。当进行insert或push_back等增加元素的操作时，如果此时动态数组的内存不够用，就要动态地重新分配，一般是当前大小的两倍，然后把原数组的内容拷贝过去。所以，在一般情况下，其访问速度同一般数组，只有在重新分配发生时，其性能才会下降。例如下面的程序。

```
1    #include <iostream>
2    #include <vector>
3    using namespace std;
4
```

```
5  int main()
6  {
7      vector<int> v;        //初始时无元素，容量为0
8      cout << v.capacity() << endl;
9      v.push_back(1)    ;    //容量不够，分配1个元素内存
10     cout << v.capacity() << endl;
11     v.push_back(2);       //容量不够，分配2个元素内存
12     cout << v.capacity() << endl;
13     v.push_back(3);       //容量不够，分配4个元素内存
14     cout << v.capacity() << endl;
15     v.push_back(4);
16     v.push_back(5);       //容量不够，分配8个元素内存
17     cout << v.capacity() << endl;
18     v.push_back(6);
19     v.push_back(7);
20     v.push_back(8);
21     v.push_back(9);       //容量不够，分配16个元素内存
22     cout << v.capacity() << endl;
23
24     return 0;
25  }
```

下面是各个执行步骤。

（1）代码第7行，初始化时v无元素（size为0），且容量（capacity）也为0。

（2）代码第9行，在数组末尾添加元素1，由于容量不够，因此allocator分配1个int大小的内存，并把整数1复制到这个内存中。

（3）代码第11行，在数组末尾添加元素2。此时容量为1，但元素

个数需要变为2，因此容量不够。于是allocator先分配原来容量的2倍大小的内存（2个int大小的内存），然后把原来数组中的1和新加入的2拷贝到新分配的内存中，最后释放原来数组的内存。

（4）代码第13行，在数组末尾添加元素3。此时容量为2，而元素个数需要变为3，因此容量也不够。和（3）相同，allocator分配4个int的内存，然后把原来数组中的1、2以及新加入的3拷贝到新分配的内存，最后释放原数组的内存。

（5）代码第15行，在数组末尾添加元素3。此时容量为4，而元素个数需要变为3，因此容量足够，不需要分配内存，直接把4拷贝到数组的最后即可。

以后的操作不再赘述。注意，vector的size()和capacity()是不同的，前者表示数组中元素的多少，后者表示数组有多大的容量。由上面的分析可以看出，使用 vector 的时候需要注意内存的使用。如果频繁地进行内存的重新分配，会导致效率低下。

【答案】

vector内部是使用动态数组的方式实现的。如果动态数组的内存不够用，就要动态地重新分配，一般是当前大小的两倍，然后把原数组的内容拷贝过去。所以，在一般情况下，其访问速度同一般数组，只有在重新分配发生时，其性能才会下降。它的内部使用allocator类进行内存管理，程序员不需要自己操作内存。

面试题3 看代码回答问题——vector容器中iterator的使用

考点：vector中iterator的使用

出现频率：★★★★

```
1   vector <int> array;
2   array.push_back( 1 );
3   array.push_back( 2 );
4   array.push_back( 3 );
5   for(vector<int>::size_type i=array.size()-1; i>=0; --i ) //反向遍历array数组
6   {
7       cout << array[i] << endl;
8   }
```

当运行代码时，没有输出 3 2 1，而是输出了一大堆很大的数字，为什么？

于是修改代码：

```
1   for(vector <int>::size_type j=array.size();j>0;j--)
2   {
3       cout << "element is " <<array[j-1] <<endl;
4   }
```

这样就输出了3 2 1，到底是为什么呢？

【解析】

由于vector支持随机访问，并且重载了[]运算符，因此可以像数组那样（比如a[i]）来访问vector中的第i+1个元素。

现在来简单分析vector中size_type的定义过程。为方便起见，下面省略了某些头文件。

在 vector定义中可以看到：

1　typedef _A::size_type size_type;

而_A是 allocator<_Ty>，因此查看 allocator的定义，不难发现：

1　typedef _SIZT size_type;

而_SIZT的定义为：

1　#define _SIZT size_t

最后 size_t的定义为：

1　typedef unsigned int size_t;

因此最后的结果为：size_type 是个unsigned int 类型成员。我们知道，无符号的整数是大于等于0的，因此上面第1段代码（代码第5行）中的i>=0永远为true，程序一直循环，输出很多随机数，最后程序崩溃。而第2段代码（代码第1行）使用了i>0作为循环的条件，即i为0时结束循环。

面试题4 看代码找错——vector容器的使用

考点：理解vector容器的使用

出现频率：★★★★

```
1    typedef vector IntArray;
2    IntArray array;
3    array.push_back( 1 );
4    array.push_back( 2 );
5    array.push_back( 2 );
6    array.push_back( 3 );
7    //删除array数组中所有的2
8    for( IntArray::iterator itor=array.begin(); itor!=array.end(); ++itor )
9    {
10       if( 2 == *itor )
11           array.erase( itor );
12   }
```

【解析】

这道题有两个错误。

（1）代码第 1 行中没有使用类型参数，这将会导致编译错误。由于 array 需要添加 int类型的元素，因此代码第1行定义vector时应该加上int类型。

```
1    typedef vector<int> IntArray;
```

（2）代码第8～12行的for循环，这里只能删除array数组中的第一个2，而不能删除所有的 2。这是因为，每次调用“array.erase(itor);”后，被删除元素之后的内容会自动往前移，导致迭代漏项，应在删除

一项后使itor--，使之从已经前移的下一个元素起继续遍历。

正确的程序如下。

```
1   #include <iostream>
2   #include <vector>
3   using namespace std;
4
5   int main()
6   {
7       typedef vector<int> IntArray;
8       IntArray array;
9       array.push_back( 1 );
10      array.push_back( 2 );
11      array.push_back( 2 );
12      array.push_back( 3 );
13      //删除array数组中所有的2
14      for( IntArray::iterator itor=array.begin(); itor!=array.end();
++itor )
15      {
16          if( 2 == *itor )
17          {
18                  array.erase( itor );
19                  --itor;    //删除一项后使itor--
20          }
21      }
22
23      //测试删除后array中的内容
24      for(itor=array.begin(); itor!=array.end(); ++itor)
```

```
25        {
26            cout << *itor << endl;
27        }
28    }
```

执行结果为：

```
1   1
2   3
```

【答案】

（1）没有使用类型参数，会导致编译错误。

（2）只能删除 array 数组中的第一个 2，而不能删除所有的 2。因为每次调用“array.erase(itor);”后，被删除元素之后的内容会自动往前移，导致迭代漏项。

面试题5 把一个文件中的整数排序后输出到另一个文件中

考点：运用vector容器解决实际问题

出现频率：★★★★

【解析】

这个题目涉及文件操作以及排序。我们可以使用 vector 容器来简化文件操作。在读文件的时候用push_back把所有的整数放入一个vector<int>对象中，在写文件时用[]操作符直接把vector<int>对象输出到文件。代码如下。

```
1    #include<iostream>
2    #include<fstream>
3    #include <vector>
4    using namespace std;
5
6    //对data容器中的所有元素进行冒泡排序
7    void Order(vector<int>& data)
8    {
9       int count = data.size();          //获得vector中的元素个数
10       for (int i=0 ; i<count ; i++)
11       {
12          for (int j=0; j<count-i-1; j++)
13          {
14              if (data[j] > data[j+1])   //如果当前元素比下一个元素
大，则交换
```

```
15              {                 //结果为升序排列
16                  int temp = data[j] ;
17                  data[j] = data[j+1] ;
18                  data[j+1] = temp ;
19              }
20          }
21      }
22  }
23
24  int main( void )
25  {
26      vector<int>data;
27      ifstream in("c:\\data.txt");
28      if (!in)                  //打开输出文件失败
29      {
30          cout<< "infile error!" << endl;
31          return 1;
32      }
33      int temp;
34      while (!in.eof())
35      {
36          in >> temp;           //从文件中读取整数
37          data.push_back(temp);   //把读取的证书放入data容器中
38      }
39      in.close();
40      Order(data);          //冒泡排序
41      ofstream out("c:\\result.txt");
```

```
42      if (!out)              //打开输出文件失败
43      {
44          cout<<"outfile error!" << endl;
45          return 1;
46      }
47      for (int i = 0 ; i < data.size() ; i++)
48          out << data[i] << " ";    //把data容器中的所有元素输出至文件
49      out.close();
50      return 0;
51  }
```

程序中的 Order 函数使用冒泡排序把 data 容器中的所有元素进行了升序。下面说明主函数的各个步骤。

（1）代码第26行，定义了一个空的vector<int>容器。

（2）代码第27～32行，定义了一个输入文件流，如果打开输入文件（data.txt）失败，则主函数返回。

（3）代码第34～38行，把输入文件（data.txt）中的所有整数放入vector容器中。

（4）代码第40行，调用Order函数对vector容器中的所有元素进行排序（升序）。

（5）代码第41～46行，定义了一个输出文件流，如果打开输出文件（result.txt）失败，则主函数返回。

（6）代码第47～48行，把data容器中的所有元素输出至result.txt中。

面试题6 list和vector有什么区别

考点：理解list和vector的区别

出现频率：★★★★

【解析】

vector 和数组类似，它拥有一段连续的内存空间，并且起始地址不变，因此它能非常好地支持随机存取（使用[]操作符访问其中的元素）。但由于它的内存空间是连续的，所以在中间进行插入和删除操作会造成内存块的拷贝（复杂度是 O(n)）。另外，当该数组后的内存空间不够时，需要重新申请一块足够大的内存并进行内存的拷贝。这些都大大影响了vector的效率。

list是由数据结构中的双向链表实现的，因此它的内存空间可以是不连续的。因此只能通过指针来进行数据的访问。这个特点使得它的随机存取变得非常没有效率，需要遍历中间的元素，搜索复杂度O(n)，因此它没有提供[]操作符的重载。但由于链表的特点，它可以以很好的效率支持任意地方的删除和插入。

由于list和vector上面的这些区别，list::iterator与vector::iterator也有一些不同。请看下面的例子。

```
1   #include <iostream>
2   #include <vector>
3   #include <list>
4   using namespace std;
5
6   int main( void )
7   {
```

```
8       vector<int> v;
9       list<int> l;
10
11        for (int i=0; i<8; i++)              //往v和l中分别添加元素
12        {
13            v.push_back(i);
14            l.push_back(i);
15        }
16
17        cout << "v[2] = " << v[2] << endl;
18        //cout << "l[2] = " << l[2] << endl;       //编译错误,list没有重载
[]
19        cout << (v.begin() < v.end()) << endl;
20        //cout << (l.begin() < l.end()) << endl;    //编译错误,list::iterator
没有重载<或>
21        cout << *(v.begin() + 1) << endl;
22
23        vector<int>::iterator itv = v.begin();
24        list<int>::iterator itl = l.begin();
25        itv = itv + 2;
26        //itl = itl + 2;               //编译错误,list::iterator没有重载+
27        itl++;itl++;                    //list::iterator中重载了++，只能使用
++进行迭代访问
28        cout << *itv << endl;
29        cout << *itl << endl;
30
31        return 0;
```

32　}

由于 vector 拥有一段连续的内存空间，能非常好地支持随机存取，因此 vector<int>::iterator支持“+”、“+=”、“<”等操作符。

而list的内存空间可以是不连续的，它不支持随机访问，因此list<int>::iterator不支持“+”、“+=”、“<”等操作符运算。因此代码第20、26行会有编译错误。只能使用“++”进行迭代，例如代码第27行，使用两次itl++来移动itl。还有list也不支持[]运算符，因此代码第18行出现编译错误。

总之，如果需要高效的随机存取，而不在乎插入和删除的效率，就使用vector；如果需要大量的插入和删除，而不关心随机存取，则应使用list。

【答案】

vector拥有一段连续的内存空间，因此支持随机存取。如果需要高效的随机存取，而不在乎插入和删除的效率，就使用vector。

list拥有一段不连续的内存空间，因此支持随机存取。如果需要大量的插入和删除，而不关心随机存取，则应使用list。

面试题7 分析代码问题并修正——list和vector容器的使用

考点：理解list和vector的使用

出现频率：★★★

```
1    #include <iostream>
2    #include <vector>
3    #include <list>
4    using namespace std;
5
6    int main( void )
7    {
8        list list1;
9
10       for (int i = 0; i < 8; i ++)
11       {
12           list1.push_back(i);
13       }
14
15       for (list::iterator it = list1.begin(); it != list1.end(); ++it)
16       {
17           if (*it % 2 == 0)
18           {
19                   list1.erase(it);
20           }
```

```
21        }
22
23        return 0;
24    }
```

【解析】

本题有下面两个方面的问题。

第一个问题是list1以及它的iterator都缺少类型参数，代码第8行和第15行都有编译错误。由于在代码第12行添加的元素类型为int，因此需要将第8行和第15行中的list::改成list<int>::。

第二个问题是当改正了上面编译的错误后，运行程序会导致程序崩溃。因为当第一次执行代码第19行（调用erase方法删除元素）后，it原来指向的元素内存已经被释放掉了，因此进入下一次循环后，获得it的值就出现了访问违规。

这里要注意：vector使用的是数组方式，当删除一个元素后，此元素的后面元素会自动往前移动。而list由于使用链表结构，因此执行erase时会释放链表的节点内存。但是可以通过erase的返回值获得原来链表的下一个元素。

改正后的代码如下。

```
1    #include <iostream>
2    #include <vector>
3    #include <list>
4    using namespace std;
5
6    int main( void )
7    {
8        list<int> list1;            //指定存放int类型的元素
9
```

```
10        for (int i = 0; i < 8; i ++)
11        {
12            list1.push_back(i);    //把整数放入list1中
13        }
14
15        //iterator也需要指明是int类型元素的迭代器
16        for (list<int>::iterator it = list1.begin(); it != list1.end(); ++it)
17        {
18            if (*it % 2 == 0)
19            {
20                    it = list1.erase(it);    //得到下一个元素的位置
21                    it--;                    //回到上一个元素位置
22            }
23        }
24
25        for (it = list1.begin(); it!=list1.end(); ++it)
26        {
27            cout << *it << endl;    //输出list1内的各个元素
28        }
29
30        return 0;
31    }
```

执行了代码第10～13行，0～8这9个数字被放入list1中。这里简单分析一下删除过程。

当it指向第一个元素，即0时，由于0是2的倍数，因此18行的if判断为真，执行删除操作，it返回下一个元素位置，即指向原来的1。为了让进入下一次循环时it还指向1，需要让it--（不能使用it=it-1）。

最后第25～28行打印list1的所有元素。执行结果如下。

1 1

2 3

3 5

4 7

面试题8 stl::deque是一种什么数据类型

考点：理解deque容器的内部结构

出现频率：★★★

stl：deque是一种什么数据类型？（　　）

A．动态数组

B．链表

C．堆栈

D．树

【解析】

deque是由一段一段定量的连续空间组成的，因此是动态数组类型。

【答案】

A

面试题9 在做应用时如何选择vector和deque

vector和deque有什么区别？在做一个应用时，如果选择？

考点：理解vector和deque的区别

出现频率：★★★

【解析】

deque 称为双向队列容器，表面上与 vector 非常相似，甚至能在许多实现中直接替换vector。比较deque和vector两者的成员函数，可以发现下面两点。

（1）deque比vector多了push_front()和pop_front()两个函数，而这两个函数都是对于首部进行操作的。于是得到第一个使用deque的情况，就是当程序需要从首尾两端进行插入或删除元素操作的时候。

（2）deque中不存在capacity()和reserve()成员函数。在vector中，这两个函数分别用于获取和设置容器容量，例如下面的代码段。

```
1   int main()
2   {
3       vector<int> v;
4       v.push_back(1);
5       cout << v.capacity() << endl;
6       v.reserve(10);
7       cout << v.capacity() << endl;
8
9       return 0;
10  }
```

代码第3行，此时v的容量为0。

代码第4行，添加一个元素到末尾，此时v的容量（capacity）为1，元素个数（size）为1。

代码第6行，调用reserve设置v的容量，此时v的容量为10，元素个数仍然为1。

代码第4行，又添加一个元素到末尾，此时v的元素个数为2。由于元素个数小于容量，因此容量没有扩充，仍旧为10。

执行结果为：

1　1

2　10

因此，对于vector 来说，如果有大量的数据需要push back，则应当使用reserve()函数先设定其容量大小，否则会出现许多次容量扩充操作，导致效率很低。

而deque使用一段一段的定量内存，在进行内存扩充时也只是加一段定量内存，因此不存在容量的概念，也就没有capacity()和reserve()成员函数。

最后，在插入（insert）操作上，deque和vector有很大的不同。由于vector是一块连续的内存，所以插入的位置决定执行效率，位置越偏向数组首部，效率越低。而deque中的内存是分段连续的，因此在不同段中的插入效率都相同。

【答案】

vector和deque的不同点：内部数据管理不同。为了提高效率，vector在添加元素之前最好调用reserve()设置容量，而deque则不需要。

选择的方法：一般情况下选择vector；但当需要从首尾两端进行插入或删除元素操作的时候，应该选择deque。

面试题10 看代码找错——适配器stack和queue的使用

考点：理解适配器stack和queue的使用

出现频率：★★★★

```
1    #include <iostream>
2    #include <vector>
3    #include <queue>
4    #include <stack>
5    using namespace std;
6
7    int main()
8    {
9        stack<int, vector<int> > s;
10       queue<int, vector<int> > q;
11
12       for (int i=1; i <10; i++)
13       {
14           s.push(i);
15           q.push(i);
16       }
17
18       while(!s.empty())
19       {
20           cout << s.top() << endl;
```

```
21            s.pop();
22        }
23
24        while(!q.empty())
25        {
26            cout << q.front() << endl;
27            q.pop();
28        }
29
30        return 0;
31}
```

【解析】

代码第9～10行，使用vector分别定义了stack和queue。

代码第12～16行，把1～9放入s和q中。

代码第18～22行，循环打印s的栈顶元素，并且不断退栈，最终s变为空栈，打印的顺序为与入栈的顺序相反，即9 8 7 6 5 4 3 2 1。

代码第 24～28 行，与前面的操作方法一样，但是这里会出现编译错误。这是因为，queue是先进先出的，入队（调用push）是对队尾进行操作，由于q使用vector作为其序列容器，因此实际调用的是vector的push_back成员函数。而出队（调用pop）是对队首进行操作，此时q实际需要调用vector的pop_front成员函数，而vector没有这个pop_front成员。因此出现编译错误。

stack 是后进先出的，入栈和退栈都是对尾部进行操作，而 vector 有相应的 push_back和pop_back成员函数，因此代码第18～22行能够顺利执行。

【答案】

由于vector没有pop_front函数，因此代码第27行出现编译错误。

面试题11 举例说明set的用法

考点：理解关联容器set的使用

出现频率：★★★

【解析】

这里演示set中元素的插入、删除和遍历操作。程序示例如下。

```
1   #include <iostream>
2   #include <set>
3   #include <string>
4   using namespace std;
5
6   int main()
7   {
8       set <string> strset;
9       set <string>::iterator si;
10        strset.insert("cantaloupes");    //插入6个元素，其中有两个grapes
11        strset.insert("apple");
12        strset.insert("orange");
13        strset.insert("banana");
14        strset.insert("grapes");
15        strset.insert("grapes");
16        strset.erase(“banana”);
17        for (si=strset.begin(); si!=strset.end(); si++)
18        {
```

```
19            cout << *si << " ";    //打印set中所有元素
20        }
21        cout << endl;
22        return 0;
23    }
```

下面是示例程序的说明。

代码第10～15 行，往set 集合中分别插入6 个元素。由于字符串"grapes"重复了，因此实际插入的只有5个字符串。

代码第16 行，删除集合中内容为"banana"的元素。

代码第17～20 行，使用迭代器si 遍历set 集合。

执行结果：

```
1    1
2    apple cantaloupes grapes orange
```

面试题12 举例说明map的用法

考点：理解关联容器map的使用

出现频率：★★★

【解析】

map中存放的每一个元素都是一个键值对，下面的程序演示了常用的map操作。

```
1    #include <iostream>
2    #include <map>
3    #include <string>
4    using namespace std;
5
6    int main()
7    {
8        map<int, string> mapstring;              //键为int类型，值为string类型
9        mapstring.insert(pair<int, string>(1, "one"));      //插入4个元素
10       mapstring.insert(pair<int, string>(4, "four"));
11       mapstring.insert(pair<int, string>(3, "three"));
12       mapstring.insert(pair<int, string>(2, "two"));
13       mapstring.insert(pair<int, string>(4, "four four"));   //4已经存在，插入失败
14
15       mapstring[1] = "one one";              //1已经存在，修改键为1对应的值
```

```
16      mapstring[5] = "five";            //5不存在，添加键为5且值
为"five"的元素
17      cout << mapstring[1] << endl;        //打印键为1对应元素的值
18
19      mapstring.erase(2);               //删除键为2的元素
20      map<int, string>::iterator f = mapstring.find(2);     //查找键为2
的元素
21      if (f != mapstring.end())          //判断查找是否成功，如果成
功，则不相等
22      {
23          mapstring.erase(f);
24      }
25
26      map<int, string>::iterator it = mapstring.begin();
27      while(it != mapstring.end())        //使用迭代器遍历map中所
有元素
28      {
29          cout << (*it).first << " " << (*it).second << endl;   //打印元素
的键和值
30          it++;
31      }
32
33      return 0;
34   }
```

上面的程序中，mapstring 是存放pair<int, string>类型的元素。

插入操作，insert成员函数或使用[]操作符都可以进行插入。但它们有一点区别：当map中已经存在此键时，insert插入失败，例如代码第13

行的插入没有作用；而[]操作符则修改此键所对应的元素，例如代码第15行则修改键为1对应的值。

查找操作，代码第20行查找键为2的元素，如果查找成功，则迭代器指向键为2的元素，否则指向末尾，即mapstring.end()。

删除操作，这里演示了两种删除，和别的容器一样，可以删除迭代器指向的元素位置（或者两个迭代器的区间删除）。由于map中元素的键是唯一的，因此map也提供了以键删除的操作（如代码第19行）。

遍历操作，与其他stl容器遍历操作类似，使用迭代器对begin()和end()之间进行迭代。it指向的是都pair<int, string>元素。pair有两个成员：first和second，分别表示键（key）和值（value）。

程序输出结果：

```
1    one one
2    one one
3    three
4    four
5    five
```

可以看到，map遍历的结果是按照元素的键（key）的升序排列。这是默认情况，如果想让它们降序排列，只需要把map 和iterator 的声明都改为<int, string, greater<int> >就可以了，其中greater<int>表示按从大到小排列，并且键的类型是int。

面试题13 STL中map内部是怎么实现的

考点：理解关联容器map的内部使用

出现频率：★★★

【解析】

标准的 STL 关联容器（包括 set 和 map 以及 set 的衍生体 multiset 和 map 的衍生体multimap）的内部结构是一个平衡二叉树（balanced binary tree）。平衡二叉树有下面几种。

AVL-tree；

RB-tree；

AA-tree。

STL的底层机制都是以RB-tree（红黑树）完成的。RB-tree也是一个独立容器，但并不给外界使用。红黑树这个名字的得来就是由于树的每个结点都被着上了红色或者黑色，节点所着的颜色被用来检测树的平衡性。在对节点插入和删除的操作中，可能会被旋转来保持树的平衡性。平均和最坏情况下的插入、删除、查找时间都是O(lgn)。

一个红黑树是一棵二叉查找树，除了二叉查找树带有的一般要求外，它还具有下列的属性。

结点为红色或者黑色。

所有叶子结点都是空节点，并且被着为黑色。

如果父结点是红色的，那么两个子节点都是黑色的。

结点到其子孙节点的每条简单路径上都包含相同数目的黑色节点。

根结点是黑色的。

【答案】

map底层是以红黑树实现的。

面试题14 map和hashmap有什么区别

考点：理解关联容器map和hashmap的区别

出现频率：★★★

【答案】

有以下几点区别。

底层数据结构不同，map 是红黑树，hashmap 是哈希表。

map 的优点在于元素可以自动按照键值排序，而 hash map 的优点在于它的各项操作的平均时间复杂度接近常数。

map 属于标准的一部分，而hash map 则不是。

面试题15 什么是STL算法

考点：理解STL算法的概念和作用

出现频率：★★★★

【解析】

STL包含了大量的算法。它们巧妙地处理储存在容器中的数据。以reverse算法为例，我们只要简单使用reverse算法就能够逆置一个区间中的元素。

```
1   #include <iostream>
2   #include <vector>
3   #include <string>
4   #include <algorithm> //stl算法头文件
5   using namespace std;
6
7   int main()
8   {
9       int a[4] = {1, 2, 3, 4};
10      vector<string> v;
11      v.push_back("one");          //插入三个字符串
12      v.push_back("two");
13      v.push_back("three");
14      reverse<int [4]>(a, a+4);        //把数组a的所有元素逆置
15      reverse< vector<string>::iterator >(v.begin(), v.end());    //逆置v
中所有元素
16      for(vector<string>::iterator it=v.begin(); it!=v.end(); it++)
```

```
17        {
18            cout << *it << " ";        //输出v中元素
19        }
20        cout << endl;
21        for(int i=0; i<4; i++)
22        {
23            cout << a[i] << " ";        //输出数组a中元素
24        }
25        return 0;
26    }
```

程序执行结果：

```
1    three two one
2    4 3 2 1
```

可以看到，为了对数组 a 以及容器 v 中的所有元素进行逆置，我们都只调用了一个reverse方法。这里有几点要注意：

reverse 是个全局函数，而不是成员函数。

reverse 有两个参数，第一个参数是指向要操作的范围的头的指针，第二个参数是指向尾的指针。

reverse 操作一定范围的元素而不是仅仅操作容器，本题中对数组也进行了操作。

reverse和其他STL算法一样，它们是通用的，也就是说，reverse不仅可以用来颠倒向量的元素，也可以用来颠倒链表中元素的顺序，甚至可以对数组操作。它们实际上都是函数模板。

面试题16 分析代码功能——STL算法的使用

考点：理解STL算法的使用

出现频率：★★★

```
1    #include <iostream>
2    #include <deque>
3    #include <algorithm>
4    using namespace std;
5
6    void print(int elem)
7    {
8        cout << elem << " ";
9    }
10
11   int main()
12   {
13       deque<int> coll;
14       for (int i=1; i<=9; ++i)
15       {
16           coll.push_back(i);
17       }
18       deque<int>::iterator pos1;
19       pos1 = find(coll.begin(), coll.end(), 2);
20       deque<int>::iterator pos2;
```

```
21        pos2 = find(coll.begin(), coll.end(), 7);
22        for_each(pos1, pos2, print);
23        cout << endl;
24
25        deque<int>::reverse_iterator rpos1(pos1);
26        deque<int>::reverse_iterator rpos2(pos2);
27        for_each(rpos2, rpos1, print);
28        cout << endl;
29
30        return 0;
31    }
```

【解析】

本题涉及以下内容。

deque 容器的操作用法；

find 和for_each 两种泛型算法；

iterator和reverse_iterator的使用。

下面是程序的执行步骤。

（1）代码第14～17行，把1～9这9个数字放入coll容器中。

（2）代码第19、21行，调用find分别获得整数2和7在coll中的存放位置，即pos1指向2，pos2指向7。

（3）代码第22行，调用foreach对pos1到pos2的区间元素依次执行print，打印各个元素值。由于左闭右开的原则，从2（pos1）开始到6（pos2 前一位）结束，打印结果为：2 3 4 5 6。

（4）代码第25、26行，分别根据迭代器pos1和pos2得到反向迭代器rpos1和rpos2，即rpos1指向1，rpos2指向6。

（5）代码第27行，此时使用rpos2和rpos1反向打印各个元素值。由于左闭右开的原则，从6（rpos2）开始到2（rpos1 前一位）结束，打印

结果为：6 5 4 3 2。

【答案】

1　2 3 4 5 6

2　6 5 4 3 2

面试题17 vector中的erase方法与algorithm中的remove有什么区别

考点：理解vector中的erase方法和algorithm中的remove方法的区别

出现频率：★★

【解析】

通过下面这个程序说明两者的区别。

```
1    #include <iostream>
2    #include <vector>
3    #include <algorithm>
4    using namespace std;
5
6    template <class T>
7    void print(vector<T> &a)
8    {
9        //打印容器a中的所有元素
10       for(vector<T>::iterator it=a.begin(); it!=a.end(); it++)
11       {
12           cout << *it << " ";
13       }
14       cout << endl;
15   }
16
17   int main()
18   {
```

```
19      vector<int> array;
20
21      array.push_back(1);
22      array.push_back(2);
23      array.push_back(3);
24      array.push_back(3);
25      array.push_back(4);
26      array.push_back(5);
27
28      array.erase(array.begin());                //调用erase删除1
29      print(array);
30      vector<int>::iterator pos;
31      pos = remove(array.begin(), array.end(), 2);      //调用remove删
除2
32      print(array);
33      if ((pos + 1) == array.end())
34      {
35          cout << "(pos+1) == array.end()" << endl;
36      }
37      remove(array.begin(), array.end(), 3);         //调用remove删除
所有3
38      print(array);
39      return 0;
40   }
```

程序的执行结果如下。

```
1    2 3 3 4 5
2    3 3 4 5 5
```

3　(pos+1) == array.end()

4　4 5 5 5 5

代码第28行，调用erase删除1。由第一行打印结果可以看出，1确实被删除了，并且array中的元素个数也减了1。

代码第31行，调用erase删除2。由第2行和第3行打印结果可以看出，2被删除了，然而array中的元素个数没有改变，末尾多了一个5。

代码第37行，调用erase删除3。由于容器内有两个元素都为3，因此这两个3都被清除了，末尾多了两个5。

通过上面的分析可以得出erase和remove的区别：

vector中erase 是真正删除了元素，迭代器不能访问了。

而 algorithm 中的 remove 只是简单地把要 remove 的元素移到了容器最后面，迭代器还是可以访问到的。这是因为 algorithm 通过迭代器操作，不知道容器的内部结构，所以无法做到真正删除。

面试题18 什么是auto_ptr（STL智能指针）？如何使用

考点：理解auto_ptr智能指针的使用

出现频率：★★★★

【解析】

许多数据重要的结构以及应用，例如链表、STL 容器、串、数据库系统以及交互式应用必须使用动态内存分配，因此仍然冒着万一发生异常导致内存溢出的风险。C++标准化委员会意识到了这个漏洞并在标准库中添加了一个特殊的类模板，它就是 std::auto_ptr，其目的是促使动态内存和异常之前进行平滑的交互。auto_ptr保证当异常掷出时，分配的对象能被自动销毁，内存能被自动释放。下面是auto_ptr的用法。

```
1    #include <iostream>
2    #include <string>
3    #include <memory>
4    using namespace std;
5
6    class Test
7    {
8    public:
9        Test() {name = NULL;}
10       Test(const char* strname)                    //构造函数
11       {
12           name = new char[strlen(strname)+1];    //分配内存
13           strcpy(name, strname);                  //拷贝字符串
```

```
14      }
15      Test& operator = (const char *namestr)      //赋值函数
16      {
17          if (name != NULL)
18          {
19                  delete name;                    //释放原来的内存
20          }
21          name = new char[strlen(namestr)+1];         //分配新内存
22          strcpy(name, namestr);                      //拷贝字符串
23          return *this;
24      }
25      void ShowName() {cout << name << endl;}         //打印name
26    ～Test()                                  //析构函数
27      {
28          if (name != NULL)
29          {
30                  delete name;                        //释放name所指内存
31          }
32          name = NULL;
33          cout << "delete name" << endl;
34      }
35  public:
36      char *name;
37  };
38
39  int main()
40  {
```

```
41    auto_ptr<Test> TestPtr(new Test("Terry"));    //TestPtr智能指
针
42    //auto_ptr<Test> TestPtr = new Test("Terry");    //编译错误
43    TestPtr->ShowName();         //使用智能指针调用
ShowName()方法
44    *TestPtr = "David";          //使用智能指针改变字符串内
容
45    TestPtr->ShowName();         //使用智能指针调用
ShowName()方法
46
47    int y = 1;
48    int x = 0;
49    y = y / x;                   //产生异常,TestPtr指向对象的内存
仍然能得到释放
50    return 0;
51  }
```

在这里，我们定义了一个Test类用于测试。Test类有一个成员name，并实现它的构造函数、赋值函数以及析构函数，另外还有ShowName()打印name字符串。下面是使用auto_ptr操作的步骤。

代码第37行，创建并初始化auto_ptr。auto_ptr后面的尖括弧里指定auto_ptr指针的类型，在这个例子中是Test。然后auto_ptr句柄的名字，在这个例子中是TestPtr。最后是用动态分配的对象指针初始化这个实例。注意，只能使用auto_ptr构造器的拷贝，也就是说，代码第38行使用赋值的方式是非法的，因此第38行会出现编译错误。

代码第39行，使用TestPtr调用Test中的ShowName()成员函数。和一般指针操作相同，这里使用的是->操作符。

代码第40行，使用TestPtr对原对象进行赋值。和一般指针相同，这

里也是使用*操作符。

代码第41行，再次调用ShowName()打印name字符串。

代码第44行，这里会发生除0的异常，程序崩溃，但是智能指针指向的内存仍然能得到释放。

没有注释代码第44行，即程序不会崩溃，则main函数返回前，TestPt发生析构，这时会调用Test的析构函数。

程序执行结果：

1 Terry

2 David

3 程序崩溃

4 Delete name

由此可以看出，使用auto_ptr可以代替指针进行类似指针的操作，并且不用关心内存释放。auto_ptr 是如何解决前面提到的内存溢出问题的呢？auto_ptr 的析构函数自动摧毁它绑定的动态分配对象。也就是说，当 TestPt 的析构函数执行时，它删除构造 TestPt 期间创建的Test对象指针。

面试题19 看代码找错——智能指针auto_ptr的使用

考点：理解智能指针auto_ptr的使用

出现频率：★★★

下面的语句有什么错误？

```
std::auto_ptr ptr(new char[10]);
```

【答案】

auto_ptr后面的尖括弧里必须指定auto_ptr指针的类型，这里为char。正确的语句为：

```
std::auto_ptr<char> ptr(new char[10]);
```

面试题20 智能指针如何实现

考点：理解智能指针的实现细节

出现频率：★★

【解析】

智能指针是用来实现指针指向的对象的共享的。其实现的基本思想：

每次创建类的新对象时，初始化指针并将引用计数置为1；

当对象作为另一对象的副本而创建时，拷贝构造函数拷贝指针并增加与之相应的引用计数；

对一个对象进行赋值时，赋值操作符减少左操作数所指对象的引用计数（如果引用计数减至0，则删除对象），并增加右操作数所指对象的引用计数；

调用析构函数时，减少引用计数（如果引用计数减至0，则删除基础对象）；

重载“->”以及“*”操作符，使得智能指针有类似于普通指针的操作。

根据以上分析，首先可以得出下面的类模板原型。

```
1    template <class T>
2    class SmartPtr
3    {
4    public:
5        SmartPtr(T *p = 0);                          //构造函数
6        SmartPtr(const SmartPtr& src);               //拷贝构造函数
7        SmartPtr& operator = (const SmartPtr& rhs);  //赋值函数
8        T* operator -> ();                           //重载->
```

```
9     T& operator * ();                    //重载*
10   ～SmartPtr();                        //析构函数
11   private:
12      void decrRef()
13      {                                //被其他成员函数所调用
14          if (--*m_pRef == 0)                //自身的引用计数减1
15          {                            //如果计数为0，则释放内存
16                  delete m_ptr;
17                  delete m_pRef;
18          }
19      }
20      T *m_ptr;                          //保存对象指针
21      size_t *m_pRef;                       //保存引用计数
22   };
```

上面的私有成员函数decrRef将引用计数减1。如果引用计数减至0，则删除m_ptr所指对象。根据前面的分析，decrRef只被赋值函数以及析构函数使用。

下面说明各个成员的具体定义。

首先是构造函数与析构函数的定义。普通构造函数中，m_ptr与p指向同一块内存，并初始化引用计数为1。拷贝构造函数中与普通构造函数的不同之处为引用计数需要加1。析构函数调用私有成员decrRef对引用计数递减，并且判断是否需要释放对象。代码如下。

```
1   template<class T>
2   SmartPtr<T>::SmartPtr(T *p)       //普通构造函数
3   {
4     m_ptr = p;             //m_ptr与p指向同一内存
5     m_pRef = new size_t(1);    //m_pRef初值为1
```

```
6    }
7
8    template<class T>
9    SmartPtr<T>::SmartPtr(const SmartPtr<T>& src)        //拷贝构造函数
10   {
11       m_ptr = src.m_ptr;        //m_ptr与src.m_ptr指向同一内存
12       m_pRef++;
13       m_pRef = src.m_pRef;        //拷贝引用
14   }
15
16   template<class T>
17   SmartPtr<T>::～SmartPtr()        //析构函数
18   {
19       decrRef();                        //引用减1,如果减后的引用为0，则释放内存
20       std::cout<<"SmartPtr: Destructor"<<std::endl;
21   }
```

接下来是“->”和“*”的重载。这两个函数很简单，只需要分别返回m_ptr以及m_ptr所指的内容即可。注意，如果m_ptr此时为空，则应该抛出异常。代码如下。

```
1    template<class T>
2    T* SmartPtr<T>::operator -> ()    //重载 ->
3    {
4        if (m_ptr)
5                return m_ptr;
6        //m_ptr为NULL时，抛出异常
```

```
7       throw std::runtime_error("access through NULL pointer");
8    }
9    template<class T>
10   T& SmartPtr<T>::operator * ()    //重载 *
11   {
12       if (m_ptr)
13           return *m_ptr;
14       //m_ptr为NULL时，抛出异常
15       throw std::runtime_error("dereference of NULL pointer");
16   }
```

最后是赋值函数的实现：

```
1    template<class T>
2    SmartPtr<T>& SmartPtr<T>::operator = (const SmartPtr<T>&
rhs)    //赋值函数
3    {
4       ++*rhs.m_pRef;           //rhs的引用加1
5       decrRef();           //自身指向的原指针的引用减1
6       m_ptr = rhs.m_ptr;       //m_ptr合rhs.m_ptr指向同一个对象
7       m_pRef = rhs.m_pRef;    //复制引用
8       return *this;
9    }
```

这样，就可以像 std::auto_ptr那样来使用SmartPtr。测试程序如下。

```
1    int main()
2    {
3       SmartPtr<Test> t1;       //空指针
4       SmartPtr<Test> t2(new Test("Terry"));
5       SmartPtr<Test> t3(new Test("John"));
```

```
6      try
7      {
8            t1->ShowName()   ;   //空指针调用抛出异常
9      } catch (const exception& err)
10       {
11           cout << err.what() << endl;
12   }
13       t2->ShowName();         //使用t2调用showName()
14       *t2 = "David";          //使用t2给对象赋值
15       t2->ShowName();         //使用t2调用showName()
16       t2 = t3;                //赋值，原来t2的对象引用为0，发生析构
17                               //而t3的对象引用加1
18       cout << "End of main..." << endl;
19       return 0;
20   }
```

main函数代码第8行，t1指向一个NULL指针，因此调用ShowName时会出现异常（异常在重载的“->”函数中被抛出）。

main函数代码第12～15行，使用SmartPtr对象对Test对象进行操作，其方法与使用Test对象指针的操作方法相同。

main函数代码第16行，对t2进行赋值操作，操作完成后，t2引用的原对象发生析构（此对象没有SmartPtr对象引用了），t2和t3引用同一个对象，于是这个对象的引用计数加1。注意，这里我们并没有显示地对t2所引用的原对象进行释放操作，这就是智能指针的精髓所在。

面试题21 使用std::auto_ptr有什么方面的限制

考点：理解智能指针auto_ptr的使用限制

出现频率：★★

【解析】

标准C++提供的auto_ptr。而auto_ptr的使用是有很多限制的，我们一条一条来细数。

（1）std::auto_ptr 要求一个对象只能有一个拥有者，严禁一物二主，这一点和前面实现的SmartPtr不同。比如以下用法是错误的。

```
1    classA *pA = new classA;
2    auto_ptr<classA> ptr1(pA);
3    auto_ptr<classA> ptr2(pA);
```

（2）std::auto_ptr是不能以传值方式进行传递的。

因为所有权的转移，会导致传入的智能指针失去对指针的所有权。如果要传递，可以采用引用方式，利用const引用方式还可以避免程序内其他方式的所有权的转移。

（3）其他注意事项：

不支持数组。

注意其Release 语意。Release 是指释放出指针，即交出指针的所有权。

auto_ptr在拷贝构造和=操作符时的特殊含义决定了它不能做为STL标准容器的成员。

面试题22 如何理解函数对象

考点：理解函数对象的概念

出现频率：★★★

【解析】

简单地说，函数对象就是一个重载了“()”运算符的类的对象，它可以像一个函数一样使用。例如

```
1   #include <iostream>
2   using namespace std;
3
4   class MyAdd
5   {
6   public:
7       int operator () (int a, int b) { return a+b; }
8   };
9
10  class MyMinus
11  {
12  public:
13      int operator () (int a, int b) { return a-b; }
14  };
15
16  int main()
17  {
18      int a = 1;
```

```
19      int b = 2;
20      MyAdd addObj;
21      MyMinus minusObj;
22      cout << "a+b=" << addObj(a, b) << endl;       //输出 a+b=3
23      cout << "a-b=" << minusObj(a, b) << endl;   //输出a-b=-1
24      return 0;
25   }
```

可以看出，由于类MyAdd和类MyMinus都重载了“()”运算符，因此它们的对象addObj和minusObj可以像函数那样使用。

STL中提供了一元和二元函数的两种函数对象。下面列出了各个函数对象。

一元函数对象：

negate，相反数。

二元函数对象：

plus，加（+）法。

minus，减（-）法。

multiplies，乘（*）法。

divides，除（/）法。

modulus，求余（%）。

equal_to，等于（==）。

not_equal_to，不等于（!=）。

greater，大于（>）。

greater_equal，大于或等于（>=）。

less，小于（<）。

less_equal，小于或等于（<=）。

logical_and，逻辑与（&&）。

logical_or，逻辑或（||）。

logical_not，逻辑非（!）。

以上这些函数对象都是基于模板实现的，可以这样使用它们：

```
1    minus <int> int_ minus;
2    cout << int_minus(3, 4) << endl; //输出-1
```

【答案】

函数对象就是一个重载了“()”运算符的类的对象，它可以像一个函数一样使用。

面试题23 如何使用bind1st和bind2nd

考点：理解bind1st和bind2nd的使用

出现频率：★★★

【解析】

bind1st 和 bind2nd 都是函数适配器，用于将二元函数对象转换为一元函数对象。下面说明bind1st的用法。例子程序如下。

```
1    #include <iostream>
2    #include <algorithm>
3    #include <functional>
4    #include <vector>
5    using namespace std;
6
7    int main()
8    {
9        //plus的第一个参数为10
10       binder1st<plus<int> > plusObj = bind1st(plus<int>(), 10);
11       //minus的第一个参数为10
12       binder1st<minus<int> > minusObj = bind1st(minus<int>(), 10);
13
14       cout << plusObj(20) << endl;    //执行10 + 20，打印30
15       cout << minusObj(20) << endl;    //执行10 - 20，打印10
16
17       vector<int> v;
18       for (int i=1; i<10; i++)
```

```
19      {
20          v.push_back(i); //v: 1,2,3,4,5,6,7,8,9
21      }
22
23      //使用count_if获得v中大于或等于4的个数
24      int n = count_if(v.begin(), v.end(), bind1st(less_equal<int>(), 4));
25      cout << "大于或等于4的数有" << n << "个。" << endl;
26
27      return 0;
28  }
```

代码第9～12行，使用bind1st对标准库中的plus和minus函数分别进行装配。因为plus 和 minus 都是二元函数对象，也就是说它们都有两个参数，所以经过装配后，plusObj和minusObj都变成了一元函数对象（只有1个参数），bind1st<plus<int>>和binder1st<minus <int>>分别是plusObj和minusObj的类型。binder1st表示绑定的是第一个参数（bind first），因此plusObj(20)执行的是10+20，而minusObj(20)执行的是10-20，这里的10都是被bind1st绑定的参数。

STL中可以利用bind1st、bind2nd使得算法一般化，以count_if为例。代码第24行， bind1st(less_equal<int>(), 4)返回的是一个用于与 4 比较的一元函数对象。注意比较时 4 在“<=”左边，vector的各个元素值在右边，因此得出的是vector中大于或等于4的元素个数。

程序执行结果：

```
1   30
2   -10
3   大于或等于4的数有6个
```

bind2nd与bind1st用法类似，只有一点不同，就是它绑定的是第二

个参数。因此，如果要使用bind2nd完成上面代码第24行的运算，只需要把less_equal换成greater_equal就可以了，即：

```
1    int n = count_if(v.begin(), v.end(), bind2nd(greater_equal<int>(), 4));
```

【答案】

bind1st和bind2nd都是函数适配器，用于将二元函数对象转换为一元函数对象。bind1st绑定的是第一个参数，而bind2nd绑定的是第二个参数。

面试题24 实现bind1st的函数配接器

考点：理解函数配接器的实现

出现频率：★★★

标准 C++模板库中有一个名为 bind1st 的函数配接器（实际就是一个函数模板）。它接受两个参数，一个是二元函数对象bin_op，一个是二元函数对象的参数value。返回一个新的一元函数对象uni_op。使用uni_op(param)等效于bin_op(value,param)，即二元函数对象的第一个value被“固定”了。

试编写程序实现一个类似功能的my_bind1st函数配接器，并给出相应的测试代码。

【解析】

程序代码如下。

```
1   #include <iostream>
2   #include <algorithm>
3   #include <functional>
4   using namespace std;
5
6   template <class Operation, class Param>
7   class my_binder1st
8   {
9   public:
10      my_binder1st(Operation op, Param first)
11      {
12          m_op = op;                    //二元函数对象赋值
```

```
13          m_first = first;             //绑定的参数赋值
14      }
15      Param operator () (Param second)    //重载()运算符
16      {
17          return m_op(m_first, second);    //返回二元函数对象执行结
果
18      }
19  private:
20      Operation m_op;                      //二元函数对象
21      Param m_first;                       //绑定的第一个参数
22  };
23
24  template <class Operation, class Param>
25  my_binder1st<Operation, Param>
26  my_bind1st(Operation op, Param first)
27  {
28      //返回一个新的一元函数对象
29      return my_binder1st<Operation, Param>(op, first);
30  }
31
32  int main()
33  {
34      //plus的第一个参数为10
35      my_binder1st<plus<int>, int> plusObj = my_bind1st(plus<int>
(), 10);
36      //minus的第一个参数为10
37      my_binder1st<minus<int>, int> minusObj =
```

my_bind1st(minus<int>(), 10);

```
38
39      cout << plusObj(20) << endl;      //执行10 + 20，打印30
40      cout << minusObj(20) << endl;      //执行10 - 20，打印10
41
42      return 0;
43  }
```

程序中类模板 my_binder1st 重载了“()”运算符并且只有一个参数second（代码第 17行），因此它是一个一元函数对象。my_binder1st有两个私有成员，一个是二元函数对象op，另一个是绑定的第一个参数first。这两个私有成员是通过构造函数被赋值的。

函数模板my_bind1st用来得到一个my_binder1st模板类对象，通过把二元函数对象以及绑定的first参数传入，得到转换之后的一元函数对象。

最后main函数中对my_bind1st进行了测试。从代码第39行和第40行的打印结果可以看出，plusObj和minusObj确实转换成功。

第12章 智力测试题

有很多有趣的逻辑思考题目出现于跨国企业的招聘面试中，它对考查一个人的思维方式及思维方式的转变能力有极其明显的作用。据一些研究显示，这样的能力往往也与工作中的应变与创新状态息息相关。

这类智力型题目看似稀奇古怪。其实一般来说，它们并不是真要考你的智力，而是考以下几点。

思维的方式。这类问题的典型就是微软的“美国有多少个加油站”。应对：此类问题，其实并没有唯一正确的答案，面试者希望看到的是应聘者在分析和解决此问题中展示出来的思维过程。应对此类问题，注意力应放在合理假设、正确推理上，尽量把自己清晰的思维过程让面试者完全明白，而非什么数字是正确的。

逻辑推理能力。这类问题的典型就是德勤的“谁是养猫人”。一般都会给出一系列条件，然后让你从相互关系中推出答案。应对：逻辑关系可能较复杂，建议拿纸写下推理过程，清晰地表达自己的思路。即使最终没能得出正确答案，也能让面试者知道你是怎么推理的。准备时应找几个典型例子，推导几遍，基本套路都一样。

处事应变能力。此类问题中很多类似大家常见的脑筋急转弯，有的还令人尴尬。一部分问题超常得奇怪，只有平时多注意积累；有些无标准答案，冷静处理即可。

所以，回答这些题目时，必须打破思维定式，试着从不同的角度考虑问题，不断进行逆向思维、换位思考，并且把题目与自己熟悉的场景联系起来，切忌思路混乱。最重要的是要尽量让面试者知道你到底是怎

么得出你的结论的。至于具体答案是否靠谱，反而是第二位的。

面试题1 元帅领兵

元帅统领8员将，每将各分8个营，每营里面摆8阵，每阵配置8先锋，每个先锋8旗头，每个旗头有8队，每队分设8个组，每组带领8个兵。请你掐指算一算，元帅共有多少兵？

【解析】

首先，元帅统领8员将，每将各分8个营，即元帅统领8×8个营；

接下来，每营里面摆8阵，即元帅统领8×8×8个阵。

以下的计算和上面类似，即不断地乘以一个整数。

由于8个条件都是数字8，所以元帅总共有8×8×8×8×8×8×8×8个兵。

【答案】

元帅总共有8×8×8×8×8×8×8×8个兵。

面试题2 两龟赛跑

有两只乌龟一起赛跑。甲龟到达10米终点线时，乙龟才跑到9米。现在如果让甲龟的起跑线退后1米，这时两龟再同时起跑比赛，问：甲、乙两龟是否同时到达终点？

【解析】

这道题有一个陷阱。如果我们不仔细思考，仅仅从距离方面来判断，则会很容易得出甲、乙两龟同时到达终点的错误判断。

我们知道，甲、乙两龟各自的用时等于各自的距离除以各自的速度。由前面的条件，也就是甲龟到达10米终点线时，乙龟才跑到9米可知，甲龟的速度是乙龟的10/9倍。设甲龟的速度为每小时v米，则乙龟的速度为每小时0.9v米。

如果让甲龟的起跑线退后1米，则甲龟一共跑11米，而乙龟仍然是10米，则甲龟跑完11米需11/v时，而乙龟跑完10米需10v/9时。显然10v/9大于11/v，所以乙龟跑完10米比甲龟跑完11米用时要长，即甲龟先到。

【答案】

甲龟先到终点。

面试题3 电视机的价格

麦克因工作繁忙，决定临时请尼克来协助他工作。规定以一年为期限，一年的报酬为600美元与一台电视机。

可是尼克做了7个月后，因急事必须离开麦克，并要求麦克付给他应得的钱和电视机。由于电视机不能拆散付给他，结果尼克得到了150美元和一台电视机。

现在请你想一想：这台电视机值多少钱？

【解析】

根据题意，我们可以得到下面两个条件。

（1）尼克一年的报酬为600美元与一台电视机。

（2）尼克工作7个月后得到了150美元和一台电视机。

假设尼克一个月的报酬是x美元，并且这台电视机的价值是y美元，则我们可以把上面两个条件转换成一个二元一次方程组，通过解方程组得到电视机的价值。

方程组为：

1 $12x = 600 + y$;

2 $7x = 150 + y$;

方程组的解为：

1 $x = 90$

2 $y = 480$

因此，电视机值480美元。

【答案】

电视机值480美元。

面试题4 这块石头究竟有多重

有4个小孩看见一块石头正沿着山坡滚下来，便议论开了。

“我看这块石头有17千克重。”第1个孩子说。

“我说它有26千克。”第2个孩子不同意地说。

“我看它重21千克。”第3个孩子说。

“你们都说得不对，我看它的正确质量是20千克。”第4个孩子争着说。

他们四人争得面红耳赤，谁也不服谁。最后他们把石头拿去称了一下，结果谁也没猜准。其中一个人所猜的质量与石头的正确质量相差 2 千克，另外两个人所猜的质量与石头的正确质量之差相同。当然，这里所指的差，不考虑正负号，取绝对值。请问：这块石头究竟有多重？

【解析】

根据题意，可以整理为下面几个条件。

（1）石头的质量不为17、20、21、26中的一个。

（2）一人所猜的质量与石头的正确质量相差2千克。

（3）另外两个人所猜的质量与石头的正确质量之差相同（取绝对值）。

由条件（1）和（2）可知，石头的重量只有下面几种可能。

15千克、19千克、18千克、22千克、23千克、24千克、28千克。

为了方便叙述，我们称 4 个孩子分别为甲、乙、丙、丁。接下来对每一种可能的情况进行分析。

石头若重 15 千克，此时只有甲满足条件（2），则条件（3）中的两人可能是乙、丙、丁。由于乙、丙、丁猜的质量与石头的正确质量之差分别为 5、6、11，都不可能相同，所以与条件（3）不可能相符。

石头若重 19 千克，此时甲或乙都满足条件（2）。如果甲满足条件（2），则条件（3）中的两人可能是乙、丙、丁，由于乙、丙、丁猜的质量与石头的正确质量之差分别为1、2、7，都不可能相同，所以与条件（3）不可能相符；如果乙满足条件（2），则条件（3）中的两人可能是甲、丙、丁，由于甲、丙、丁猜的质量与石头的正确质量之差分别为1、2、7，都不可能相同，所以与条件（3）不可能相符。

由于判断原则简单，在这里对剩余可能的情况的分析不再赘述，可以推断出石头为 23千克。此时丙满足条件（2），乙和丁满足条件（3）。

【答案】

石头重23千克。

面试题5 四兄弟的年龄

一家有 4 个兄弟，他们 4 人的年龄乘起来的积为 14。那么，他们各自的年龄是多大?当然，年龄应该是整数。

【解析】

本题只有一个条件，就是4人的年龄乘起来的积为14。由于数字14只能被分解为下面两种乘积：

1 * 1 * 2 * 7

1 * 1 * 1 * 14

因此，四兄弟只能是以上两种组合。

【答案】

1、1、2、7或者1、1、1、14。

面试题6 爬楼梯

一位先生要到10层楼的第8层去办事，不巧正赶上停电，电梯无法使用，他只能够步行上楼。如果他从第1层爬到第4层需要用48秒，那么请问：以同样的速度走到第8层需要多少秒？

【解析】

注意距离的计算。因为第1层是起点，所以从第1层到第4层的距离是3个楼层。同样，从第1层爬到第8层的距离是7个楼层。根据条件可知，这位先生步行3个楼层需要48秒，则走完每个楼层需要16秒，因此可知他行7个楼层需要112秒。

【答案】

以同样的速度走到第8层需要112秒。

面试题7 3只砝码称东西

现在有3种不同质量的标准砝码：1克、3克、9克。请问：可以称出多少不同物品的质量？在进行称量时，要称的东西与已知的标准砝码可以任意地放在天平的两盘之一。另外，每种砝码都只有一只，而且不准复制。

【解析】

现在有3种不同质量的标准砝码：1克、3克、9克，显然可能称量的范围必定为1～13克。现在对于1～13克之间的各个质量进行讨论。

称量1克的物品时，天平左边放1克砝码，右边放物品。

称量2克的物品时，天平左边放3克砝码，右边放物品+1克砝码。

称量3克的物品时，天平左边放3克砝码，右边放物品。

称量4克的物品时，天平左边放1克砝码+3克砝码，右边放物品。

称量5克的物品时，天平左边放9克砝码，右边放1克砝码+3克砝码+物品。

称量6克的物品时，天平左边放9克砝码，右边放物品+3克砝码。

称量7克的物品时，天平左边放9克砝码+1克砝码，右边放物品+3克砝码。

称量8克的物品时，天平左边放9克砝码，右边放物品+1克砝码。

称量9克的物品时，天平左边放9克砝码，右边放物品。

称量10克的物品时，天平左边放1克砝码+9克砝码，右边放物品。

称量11克的物品时，天平左边放9克砝码+3克砝码，右边放物品+1克砝码。

称量12克的物品时，天平左边放9克砝码+3克砝码，右边放物品。

称量13克的物品时，天平左边放1克砝码+9克砝码+3克砝码，右边

放物品。

所以，1克到13克中任何质量的物品都能称量。

【答案】

1克到13克中任何质量的物品都能称量。

面试题8 称米

现有米9千克以及50克和200克的砝码各一个。问：怎样在天平上只称量3次而称出2千克米？

【解析】

根据题意，我们手上有50克和200克的砝码各一个，即总共是250克（0.25千克）的砝码。如果用砝码直接称量，比如

第1次使用两个砝码称出250克米；

第2次使用砝码和米称出500克米；

第3次使用所有的砝码和米称出1千克米。

按照上面这种方式称量不可能得到2千克米，难道我们有什么条件没有注意么？

此题需要注意的是已知米的初始质量为 9 千克，需要称量出来的 2 千克米与砝码质量之和为2.25千克，而2.25千克乘以4正好是9千克，于是我们可以采取如下步骤进行称量。

（1）第 1 次称量，不用砝码，直接把 9 千克米平均放到天平两端，平衡后，天平两端都称出4.5千克米。

（2）第2次称量，同第1次一样，也不用砝码，直接把4.5千克米平均放到天平两端，平衡后，天平两端都称出2.25千克米。

（3）第3次称量，把两个砝码都放在天平的一端，然后把第2次称出的2.25千克米放在天平的另一边，此时肯定是米那一端重，最后把米放入砝码端，直至两端平衡。此时，砝码端含有2千克的米。

【答案】

第1次称量使用9千克米称出4.5千克米；第2次与第1次相同，称出2.25千克米；第3次使用2.25千克米和0.25千克砝码称出2千克米。

面试题9 比萨饼交易

在我最喜欢的那家比萨饼店中，10寸的比萨卖4.99美元。店主说，他们有一笔12寸比萨饼的交易，定价为每份5.39美元。请问：该店在这笔比萨饼交易中给予了买方多少折扣？

【解析】

根据条件，10寸的比萨卖4.99美元，也就是每寸比萨卖0.499美元。那么12寸的比萨饼按规定要卖0.499×12=5.988美元。然而，这一笔12寸比萨饼的定价为每份5.39美元，也就是便宜了5.988-5.39=0.598美元，折扣为0.598/5.988，约为0.1。

【答案】

折扣约为0.1。

面试题10 伊沙贝拉时装精品屋

纽约伊沙贝拉时装精品屋新近从意大利购进了一件女式冬装。这件衣服的购入价格再加二成，是该店标出的销售价。

由于这件衣服在半个月内未卖出去，女老板又将这个定价减去了一成，于是这件衣服很快被一位漂亮的小姐买走了。女老板获利400元。

请问：这件高档女式冬装的购入价是多少？

【解析】

本题有下面几个条件。

（1）衣服的购入价格再加二成，是该店标出的销售价。

（2）定价减去了一成，很快被一位漂亮的小姐买走了。女老板获利400元。

假设衣服购入价为X元，则由条件（1）可知，原来的销售价为1.2X元。由条件（2）可获得下面的方程：

$1.2X \times 0.9 = X + 400$

解方程得到X = 5 000，即这件高档女式冬装的购入价为5 000 元。

【答案】

这件高档女式冬装的购入价为5 000 元。

面试题11 烧绳子的时间计算问题

烧一根不均匀的绳，从头烧到尾总共需要1小时。现在有若干条材质相同的绳子，问：如何用烧绳的方法来计时1时15分呢？

【解析】

由于绳子是不均匀的，所以燃烧的速度是不一样的，从一头燃烧用1 小时，那么从两头燃烧就会用30分钟。但要注意，用1/4进行燃烧就不一定是15分钟了。因此可以采用以下步骤。

（1）取A、B、C 3 条绳子，点燃A 绳的一端，同时点燃B 绳的两端。

（2）两端同时点燃的B绳燃尽用时30分钟，此时一端点燃的A绳也燃烧了30分钟，剩下的燃烧还要用30分钟。

（3）把A绳另一端点燃，这样A绳燃尽时已经过了45分钟。

（4）最后把C绳两端同时点燃，30分钟燃尽。

这样就一共用了1时15分。

【答案】

A绳从一头烧，同时B绳从两头烧。当B绳烧尽时，点燃A绳的另一头。最后C绳从两头烧，结束时即为1时15分。

面试题12 给工人的金条

你让工人为你工作 7 天，回报是一根金条。这根金条被平分成相连的 7 段，你必须在每天结束的时候给他们一段金条。如果只允许你两次把金条弄断，你如何给你的工人付费?

【解析】

由于只允许两次把金条弄断，因此金条最多能被分成 3 份。由于每天都必须付给工人金条，因此必然有1/7那一份。剩下6/7的分割，有下面几种组合。

3/7 和3/7；

1/7 和5/7；

2/7 和4/7。

对于第1种组合，在第2天就不能付给工人金条了；而对于第2种组合，在第3天就不能付给工人金条了。

因此只有第3种组合，即把金条分成1/7、2/7和4/7这3份。付费过程如下。

（1）第1天必然给他1/7；

（2）第2天给他2/7，让他找回1/7；

（3）第3天再给他1/7，加上原先的2/7就是3/7；

（4）第4天给他那块4/7，让他找回那两块1/7和2/7的金条；

（5）第5天，再给他1/7；

（6）第6天和第2天一样；

（7）第7天给他找回的那个1/7。

面试题13 被污染的药丸

你有 4 个装药丸的罐子，每个药丸都有一定的质量，被污染的药丸是没被污染的药丸的质量+1。只称量一次，如何判断哪个罐子的药被污染了？

【解析】

本题的限制是只称量一次，也就是需要一次确定是 4 个罐子中的哪一个，我们只有根据称出的总质量大小来判断。并且根据题意，每个药丸都有一定的质量，所以一定的质量是已知的了。

为了对4个药罐进行区分，必须拿出不同数量的药丸。这是因为，假如从A、B两个药罐中拿出了相同的药丸，如果被污染的药罐是A、B中的一个，则此时不能区分。

采取的称量策略为：

（1）从第1个药罐中取出1颗药丸，从第2个药罐里取出2颗药丸，第3个药罐里取出3颗药丸，第4个药灌里取出4颗药丸，一起放在电子秤上称。

（2）如果与标准质量相比重1，就是1号罐被污染；

（3）如果与标准质量相比重2，就是2号罐被污染；

（4）如果与标准质量相比重3，就是3号罐被污染；

（5）如果与标准质量相比重4，就是4号罐被污染。

【答案】

4个罐子中分别取1，2，3，4颗药丸，称出比标准质量重多少，即可判断出哪个罐子的药被污染。

面试题14 称量罐头

为罐头工厂工作的送货员A，给一家食品公司送了10箱菠萝罐头。每个罐头的质量是800克，每箱装20个。

正当他送完了货，要回工厂的时候，接到了从工厂打来的电话，说这10箱中有一箱由于机器出了问题而混进了次品，每个罐头缺50克的分量，要送货员把这箱罐头送回工厂以便更换。但是，怎样从中找出到底哪一箱是次品呢？最需要的当然是秤，可是手边又没有。

正在这时，他忽然发现不远的路旁有一台自动称量体重的机器，也就是投进去 1 元硬币就可以称量一次质量。他的口袋里刚好就有一个 1 元硬币，当然也就只能量一次。那么他应该怎么充分利用这只有一次的机会，来找到那一箱不符合规格的产品呢？

【解析】

本题与前面称药罐的面试题是同一类题，因此可以采取相同的方法。在前面的面试题中，为了对4个药罐进行区分，分别拿出了1、2、3、4颗药丸。这里为了对10箱菠萝罐头进行区分，可以分别拿出1～10个罐头，具体步骤如下。

将罐头排成一排，从左向右（反之亦然）取罐头，第1箱取1个，第2箱取2个，以此类推，第9箱取9个，第10箱取10个。全部一起过秤，若少50克，则第1箱不合格；若少100克，则第2箱不合格；依此类推，少几个50克，即为第几箱不合格。

【答案】

第1箱取1个，第2箱取两个，依此类推，第9箱取9个，第10箱取10个。全部一起过秤，若少50克，则第1箱为不合格，若少100克，则第2箱为不合格，以此类推，少个50克，即为第几箱不合格。

面试题15 有20元钱可以喝到几瓶汽水

1元钱买一瓶汽水，喝完后两个空瓶换一瓶汽水。问：你有20元钱，最多可以喝到几瓶汽水？

【解析】

根据题意，两个空瓶可以换一瓶汽水，有些人会不假思索地得出30 瓶，即 20 元钱买20瓶，喝完后这20个空瓶换10瓶汽水。但是要注意一点，使用两个空瓶得到的一瓶汽水喝完后，这个空瓶也能继续进行交换汽水瓶。

步骤如下。

（1）20元钱买了20瓶汽水，喝完后有了20个空瓶。

（2）20个空瓶换了10瓶汽水，喝完后有了10个空瓶。

（3）10个空瓶换了5瓶汽水，喝完后有了5个空瓶。

（4）5个空瓶换了2瓶汽水，喝完后有了3个空瓶。

（5）3个空瓶换了1瓶汽水，喝完后有了2个空瓶。

（6）2个空瓶换了1瓶汽水，喝完后有了1个空瓶。

（7）最后一个空瓶换1瓶汽水，喝完换来的那瓶再把瓶子还给人家即可。

这样总共有20+10+5+2+1+1+1，即40瓶汽水。

本题还有另一种解法，即由于两个空瓶可以换一瓶汽水，也就是说一个空瓶和瓶里的汽水价值相等，即都是1/2元，那么20块钱就恰恰能喝40瓶汽水。

【答案】

最多可以喝到40瓶汽水。

面试题16 判断鸟的飞行距离

有一辆火车以每小时 15 千米的速度离开北京直奔广州，同时另一辆火车每小时 20 千米的速度从广州开往北京。如果有一只鸟，以30千米/时的速度和两辆火车同时启动，从北京出发，碰到另一辆车后就向相反的方向返回去飞，就这样依次在两辆火车之间来回地飞，直到两辆火车相遇。请问：这只鸟共飞行了多长的距离？

【解析】

由于鸟的飞行速度比两列火车的速度都要快，因此从两列火车同时出发到它们相遇前，鸟会进行许多次往返飞行。如果直接计算，需要计算多次往返距离，这是一个求极限的过程。但实际上，由于鸟是以30千米/时的速度飞行的，我们只要计算它飞行了多长的时间，然后再乘以速度，就能轻易获得它飞行的距离。

假设北京和广州之间的距离是 L 千米，由于两辆火车都是匀速行驶，则它们相遇的时间为L 千米/（15 千米/时+20 千米/时）=（L/35）时。注意，两辆火车从出发到相遇的时间也是鸟飞行的时间。

鸟的速度乘以飞行的时间，即（30千米/时）×（L/35）时等于6L/7千米，也就是说，鸟总共飞行的距离是北京到广州的距离的6/7。

【答案】

鸟总共飞行的距离是北京到广州的距离的6/7。

面试题17 按劳取酬

有一个农场主，雇佣了两个临时工帮忙种小麦。其中一个叫汤姆，是一个耕地能手，但是他不会播种；而另一个叫尼克，他并不擅长于耕地，但是，他却是播种的好手。这个农场主决定要种10公顷小麦，让他们各自包一半，于是，汤姆从东头开始耕地，而尼克则从西头开始耕地。耕一亩地汤姆只要用 20 分钟，而尼克却需要 40 分钟，但是尼克播种的速度比汤姆要快3倍。

他们播种完工后，农场主按照他们的工作量给予他俩一共 100 元的工钱。请问：他们应该怎样分这份工钱才最合理？

【解析】

本题的标题是按劳取酬，也就是按照汤姆和尼克两人的工作量来分配报酬。所以要计算如何分配最后的100元，首先需要知道他们各自的总工作量。

对于两个人来说，虽然耕地速度和播种的速度各不相同，导致他们最后完成的时间不相同，但是他们各自的总工作量是一样的，即都是耕种了 5 公顷的小麦。因此按劳取酬应该每人对半分，即两人都是50元。

【答案】

两人都是50元。

面试题18 空姐分配物品

在一架飞机上，中间是一条过道，两边是座位，每一排为3人。两位空姐A和B每人负责一边，对每位旅客分配旅行物品。

开始的时候，A给右边的旅客发放了6份。此时，B过来对她说，左边应该由A负责。于是 A 重新到左边开始发放，B 接着给右边剩下的旅客发放物品，之后，又帮 A 发了 15份，最后两人同时结束工作。

请问：A和B谁发得多？多发了多少份？

【解析】

本题中虽然提到了每一排为 3 人，但是没有说总共有多少排座位，因此无法计算两位空姐A和B各自都发了多少份物品。在这里实际上没有必要计算发放的总数，由于飞机上过道的两边是一样的，因此A和B各自应发的物品总数是相同的，假设都为N份。

开始的时候，A给右边的旅客发放了6份，导致B给右边剩下的旅客发放物品。如果B发完右边的物品后没有帮A发物品，则A发了N+6份，而B发了N-6份。

但是B又帮A发了15分，导致A发给左边旅客的物品少了15份，也就是说A发了N+6-15份，即N-9份。而B发了N-6+15份，即N+9份。

显然，B发得多，B比A多发了(N+9)-(N-9)=18份物品。

【答案】

B发得多，B比A多发了18份物品。

面试题19 消失的1元钱

有3个人去住旅馆，住3间房，每一间房10元，于是他们一共付给老板30元。第2天，老板觉得3间房只需要25元就够了，于是叫服务员退回5元给3位客人。谁知服务员贪心，只退回每人1元，自己偷偷拿的2元，这样一来便等于那3位客人每人各花了9元。于是3个人一共花了27元，再加上服务员独吞的2元，总共是29元。可是当初他们3个人一共付出30元，那么还有1元在哪里呢？

【解析】

这是一道著名的偷换概念的数学题！这里服务员独吞的2元被出题者偷换了概念。

这3个人每人最后花了10-1=9元，也就是一共花了9×3=27元。

这27元包括老板得到的25元和服务员藏起的2元。如果加上他们3人每人拿回的1元×3=3元，正好是最初所付的30元。

服务员独吞的2元是包含在那27元里的，是他们付出去的钱，而不是他们拿回去的钱。本题中拿27元与服务员独吞的2元相加纯属混淆视听。

【答案】

服务员独吞的2元是包含在那27元里的，用这2元和27元相加纯属混淆视听。

面试题20 分物品

有7克、2克砝码各一个，天平一只。如何只用这些物品分3次将140克的盐分成50、90克各一份？

【解析】

这道题的意图很明显，如果只拿一个7克砝码和一个2克砝码放在天平上，称3次最多只能得到(7+2)×3=27克的盐，这与50克盐的差距很大。所以必须利用已经称出的盐去称盐。

这里给出两种称量方法。

第1种方法，3次称量的步骤为：

（1）第1次称量时，只有一个7克砝码和一个2克砝码，此时把它们放在天平的左边，右边放上盐，平衡时天平右边即可得到9克盐。

（2）第2次称量时，除了一个7克砝码和一个2克砝码之外，还有第1次称出的9克盐。把7克砝码和9克盐放在天平的左边，右边放上盐，平衡时天平右边即可得到16（7+9）克盐。

（3）第3次称量时，除了一个7克砝码和一个2克砝码之外，还有第1次称出的9克盐以及第2次称出的16克盐。把前两次称出的25（9+16）克盐放在天平的左边，右边放上盐，平衡时即可得到25克盐。此时天平左边和右边都是25克盐，总共是50克盐。

第2种方法，3次称量的步骤为：

（1）第1次称量时，只有一个7克砝码和一个2克砝码，此时把它们放在天平的左边，右边放上盐，平衡时天平右边即可得到9克盐。

（2）第2次称量时，除了一个7克砝码和一个2克砝码之外，还有第1次称出的9克盐。把所有砝码和9克盐放在天平的左边，右边放上盐，平衡时天平右边即可得到18（7+2+9）克盐。

（3）第3次称量时，除了一个7克砝码和一个2克砝码之外，还有第1次称出的9克盐以及第2次称出的18克盐。把7克砝码和第2次称出的18克盐放在天平的左边，右边放上2克砝码和盐，平衡时天平右边即可得到23克盐。总共得到9+18+23=50克盐。

上面两种方法相比，第1种方法优于第2种方法。这是因为在第2种方法的第3次称量时，其第1次称出的盐并没有放在天平的左边，所以必须找一个地方存放第1次称出的盐。

面试题21 称出4升的水

如果你有无穷多的水，一个3升的和一个5升的提桶，你如何准确称出4升的水？

【解析】

这道题可以使用倒推法。过程如下：

只有一个3升的和一个5升的提桶，因为只有5升的提桶才能装下4升的水，所以最后称出的4升的水必定在5升的桶内。

于是最后的操作只有下面两种。

5 升的桶内只有1 升的水，把满的3 升提桶的水全部倒入5 升提桶内。

3 升的桶内只有 2 升的水，把满的 5 升提桶的水往 3 升提桶内倒入1 升的水，则 5升提桶内剩余4升的水。

由于5升的提桶和3升的提桶相差2升的含水量，因此上面的第2种操作很容易就实现了。即把5升的提桶注满水，然后倒入3升的提桶中，这样，5升的提桶内只剩下2升的水了。

整理完思路，现在可以进行下面的步骤操作了。

（1）5升的桶内装满水，倒入3升的桶中，此时5升剩余2升；

（2）将3升桶中的水倒掉，将5升桶中的2升水倒入3升桶中，此时3升桶中有水2升，并且3升桶只剩余1升的空间；

（3）将5升的桶装满，然后向刚才的3升的桶中倒，使其装满，此时5升桶中剩余的水就是4升。

面试题22 通向诚实国和说谎国的路

一个岔路口分别通向诚实国和说谎国。来了两个人，已知一个是诚实国的，另一个是说谎国的。诚实国永远说实话，说谎国永远说谎话。现在你要去说谎国，但不知道应该走哪条路，需要问这两个人。你应该怎么问？

【解析】

此题的重点是我们需要分辨出这两个人中哪个人是说谎国的，哪个人是诚实国的。一旦判断出之后，我们就能顺利问路了。

根据已知条件，说谎国的人永远说假话，而诚实国的人永远说实话，我们可以设计一个答案是众所周知的问题问他们，比如以下问题。

（1）这是一个岔路口吗？

（2）你们一共是两个人吗？

（3）你们俩都是说谎国的吗？

（4）你们俩都是诚实国的吗？

对于此类问题的回答，说谎者都必然这样回答：

（1）这不是一个岔路口。

（2）我们一共不是两个人。

（3）我们俩都是说谎国的。

（4）我们俩都是诚实国的。

而诚实者的回答如下：

（1）这是一个岔路口。

（2）我们一共是两个人。

（3）我们俩不都是说谎国的。

（4）我们俩不都是诚实国的。

于是说谎者和诚实者就能被立刻分辨出来，接下来就可以顺利地问路了。

【答案】

首先设计一个答案是众所周知的问题问他们，以此分辨说谎者和诚实者，比如“你们俩都是说谎国的吗？”等问题，然后再问路。

面试题23 排序问题

有4个大小相同的球，分别为甲，乙，丙，丁。

甲和乙放在天平的一边，丙和丁在另一边，天平基本保持平衡。

乙、丙调换，乙和丁较重。

一边是甲、丁，另一边是乙，乙重。

请按重量排序！（由大到小）

【解析】

推理步骤如下。

根据3个条件，可以分别列出下面的关系式：

a．甲+乙=丙+丁

b．甲+丙<乙+丁

c．甲+丁<乙

（1）把a式和b式左、右两边分别相加后可以得出：

（甲+乙）+（甲+丙）≤（丙+丁）+（乙+丁），即甲≤丁。

（2）b式左、右两边分别减去a式左、右两边，可以得出：

（甲+丙）-（甲+乙）≤（乙+丁）-（丙+丁），即丙≤乙。

（3）把a式和c式左、右两边分别相加后可以得出：

（甲+乙）+（甲+丁）<（丙+丁）+乙，即2甲<丙。

（4）根据上面3步以及关系式c可得，排序方式可能有以下两种。

乙，丙，丁，甲；

乙，丙，甲，丁。

如果为乙，丙，甲，丁，则甲+乙>丙+丁，则不满足关系式 a，所以排序为乙，丙，丁，甲。

【答案】

重量从大到小排序为乙，丙，丁，甲。

面试题24 两个同一颜色的果冻

你有一桶果冻，其中有黄色、绿色、红色 3 种，闭上眼睛抓取同种颜色的两个。抓取多少个就可以确定你肯定有两个同一颜色的果冻？

【解析】

这道题很简单，分析步骤如下。

（1）如果只抓取了一个果冻，则其颜色是黄色、绿色、红色 3 种之中的一种，不可能存在两个同一颜色的果冻。

（2）如果抓取了两个果冻，则它们的颜色可能是黄色、绿色、红色 3 种之中的一种或两种，此时不能肯定有两个是同一颜色的。

（3）如果抓取了 3 个果冻，则它们的颜色可能是黄色、绿色、红色 3 种之中的一种、两种或3种，此时也不能肯定有两个是同一颜色的。

（4）如果抓取了 4 个果冻，则它们的颜色必然有重复，可以肯定至少有两个同一颜色的果冻。

【答案】

抓取4个就可以确定肯定有两个同一颜色的果冻。

面试题25 怎样称才能用3次就找到球

12个球、一个天平，现知道只有一个和其他的重量不同，问：怎样称才能用3次就找到那个球？

【解析】

注意，本题有一个隐含的条件，就是不知道这个小球是比其他小球轻了还是重了。正是这个条件很大程度地增加了解题的困难。

为了方便说明，把12个球分别编号为1～12。

第1次测量，先将1～4号放在左边，5～8号放在右边。此时可能会出现下面3种情况。

1．如果右边重，则坏球在1～8号之中

第2次将2～4号拿掉，将6～8号从右边移到左边，把9～11号放在右边。也就是说，把1，6，7，8放在左边，5，9，10，11放在右边。此时有（AA）、（AB）、（AC）3种情况。

（AA）如果右边重，则坏球是没有被触动的1或5号。如果是1号，则它比标准球轻；如果是5号，则它比标准球重。

第3次将1号放在左边，2号放在右边。此时又有（AAA）、（AAB）、（AAC）3种情况。

（AAA）如果右边重，则1号是坏球且比标准球轻；

（AAB）如果平衡，则5号是坏球且比标准球重；

（AAC）这次不可能左边重。

（AB）如果平衡，则坏球在被拿掉的2～4号之中，且比标准球轻。

第3次将2号放在左边，3号放在右边。此时又有（ABA）、（ABB）、（ABC）3种情况。

（ABA）如果右边重，则2号是坏球且比标准球轻；

（ABB）如果平衡，则4号是坏球且比标准球轻；

（ABC）如果左边重，则3号是坏球且比标准球轻。

（AC）如果左边重，则坏球在拿到左边的6～8号之中，且比标准球重。

第3次将6号放在左边，7号放在右边。此时又有（ACA）、（ACB）、（ACC）3种情况。

（ACA）如果右边重，则7号是坏球且比标准球重；

（ACB）如果平衡，则8号是坏球且比标准球重；

（ACC）如果左边重，则6号是坏球且比标准球重。

2．如果天平平衡，则坏球在9～12号之中

第2次将1～3号放在左边，9～11号放在右边。此时有（BA）、（BB）、（BC）3种情况。

（BA）如果右边重，则坏球在9～11号之中且坏球较重。

第3次将9号放在左边，10号放在右边。此时又有（BAA）、（BAB）、（BAC）3种情况。

（BAA）如果右边重，则10号是坏球且比标准球重；

（BAB）如果平衡，则11号是坏球且比标准球重；

（BAC）如果左边重，则9号是坏球且比标准球重。

（BB）如果平衡，则坏球为12号。

第3次将1号放在左边，12号放在右边。此时又有（BBA）、（BBB）、（BBC）3种情况。

（BBA）如果右边重，则12号是坏球且比标准球重；

（BBB）这次不可能平衡；

（BBC）如果左边重，则12号是坏球且比标准球轻。

（BC）如果左边重，则坏球在9-11号之中且坏球较轻。

第3次将9号放在左边，10号放在右边。此时又有（BCA）、

（BCB）、（BCC）3种情况。

（BCA）如果右边重，则9号是坏球且比标准球轻；

（BCB）如果平衡，则11号是坏球且比标准球轻；

（BCC）如果左边重，则10号是坏球且比标准球轻。

3．如果左边重，则坏球在1～8号之中

第2次将2～4号拿掉，将6～8号从右边移到左边，把9～11号放在右边。也就是说，把1，6，7，8放在左边，5，9，10，11放在右边。此时有（CA）、（CB）、（CC）3种情况。

（CA）如果右边重，则坏球在拿到左边的6～8号之中，且比标准球轻。

第3次将6号放在左边，7号放在右边。此时又有（CAA）、（CAB）、（CAC）3种情况。

（CAA）如果右边重，则6号是坏球且比标准球轻；

（CAB）如果平衡，则8号是坏球且比标准球轻；

（CAC）如果左边重，则7号是坏球且比标准球轻。

（CB）如果平衡，则坏球在被拿掉的2～4号之中，且比标准球重。

第3次将2号放在左边，3号放在右边。此时又有（CBA）、（CBB）、（CBC）3种情况。

（CBA）如果右边重，则3号是坏球且比标准球重；

（CBB）如果平衡，则4号是坏球且比标准球重；

（CBC）如果左边重，则2号是坏球且比标准球重。

（CC）如果左边重，则坏球是没有被触动的1或5号。如果是1号，则它比标准球重；如果是5号，则它比标准球轻。

第3次将1号放在左边，2号放在右边。此时又有（CCA）、（CCB）、（CCC）3种情况。

（CCA）这次不可能右边重。

（CCB）如果平衡，则5号是坏球且比标准球轻；
（CCC）如果左边重，则1号是坏球且比标准球重。

面试题26 计算生日是哪一天

小明和小强都是张老师的学生，张老师的生日是M月N日，2人都知道张老师的生日是下列10组中的一天。张老师把M值告诉了小明，把N值告诉了小强。张老师问他们知道他的生日是哪一天吗。

3 月4 日 3 月5 日 3 月8 日 6 月4 日 6 月7 日

9 月1 日 9 月5 日 12 月1 日 12 月2 日 12 月8 日

小明说："如果我不知道的话，小强肯定也不知道。"

小强说："本来我也不知道，但是现在我知道了。"

小明说："哦，那我也知道了。"

请根据以上对话推断出张老师的生日是哪一天，并说明原因。

【解析】

分析步骤如下。

（1）小明说："如果我不知道的话，小强肯定也不知道。"

小明能肯定小强不知道，这说明小强拿到的肯定不是7和2（因为7和2直接可以确定是6月7日和12月2日）。

小明能肯定小强拿到的不是7和2，那么他自己拿到的肯定不是6和12。

于是范围变为：

3 月4 日 3 月5 日 3 月8 日

9 月1 日 9 月5 日

（2）小强说："本来我也不知道，但是现在我知道了。"

当小强知道了小明拿到的是 3 或者 9 时，他马上就知道了准确的日期，所以小强拿到的不可能是5，只能是1、4、8中的一个。

于是范围又变为：

3 月4 日 3 月8 日

9月1日

（3）小明说：“哦，那我也知道了。”

小明知道了，这时因为月份是唯一的，即月份为9。

最后得出老师的生日为9月1日。

【答案】

老师的生日为9月1日。

面试题27 3个女儿的年龄

一个经理有3个女儿，3个女儿的年龄加起来等于13，3个女儿的年龄乘起来等于经理自己的年龄。有一个下属已知道经理的年龄，但仍不能确定经理 3 个女儿的年龄。这时经理说只有一个女儿的头发是黑的，然后这个下属就知道了经理3个女儿的年龄。请问：3个女儿的年龄分别是多少？为什么？

【解析】

分析过程如下。

（1）显然3个女儿的年龄都不为0，不然经理的年龄就为0了。

（2）因为3个女儿的年龄加起来等于13，所以可以得到下面几种组合。

1×1×11 = 11

1×2×10 = 20

1×3×9 = 27

1×4×8 = 32

1×5×7 = 35

1×6×6 = 36

2×2×9 = 36

2×3×8 = 48

2×4×7 = 42

2×5×6 = 60

3×3×7 = 63

3×4×6 = 72

3×5×5 = 75

$4 \times 4 \times 5 = 80$

（3）根据题意，有一个下属已知道经理的年龄，但仍不能确定经理 3 个女儿的年龄，这说明经理的年龄为36，因为以上的组合中，只有36的组合有两项，即1、6、6或者2、2、9。

（4）最后一个条件，经理说只有一个女儿的头发是黑的，即只有2、2、9 能够满足。所以3个女儿的年龄分别为2、2、9。

【答案】

经理3个女儿的年龄分别为2、2、9。

面试题28 取回黑袜和白袜

有两位盲人，他们都各自买了两对黑袜和两对白袜，8对袜子的布质、大小完全相同，而每对袜子都有一张商标纸连着。两位盲人不小心将 8 对袜子混在一起。他们每人怎样才能取回黑袜和白袜各两对呢？

【解析】

由于黑袜和白袜的布质、大小完全相同，所以盲人不能通过用手去摸来区别黑袜和白袜，当然也不能通过看来区别。

由于既不能通过看，也不能通过摸，只有寻找别的途径了。本题中，还有一个重要的条件，就是每对袜子都有一张商标纸连着。是不是可以通过这个条件来解决问题呢？

我们知道，袜子是两只成一双的，并且 8 对袜子有 4 双黑袜和 4 双白袜。由于是两个盲人，恰好能平均分配，因此把每双袜子的商标纸撕开，然后每人拿每双的一只，直到把 8对袜子全部分完，结果每人都拿到了 4只黑袜和 4只白袜，即两对黑袜和两对白袜。

【答案】

把每双袜子的商标纸撕开，然后每人拿每双的一只。

面试题29 谁先击完40下鼠标

击鼠标比赛现在开始!参赛者有拉尔夫、威利和保罗。拉尔夫10秒钟能击10下鼠标；威利20秒钟能击20下鼠标；保罗5秒钟能击5下鼠标。以上每人所用的时间是这样计算的：从第一击开始，到最后一击结束。

他们是否打成平手？如果不是，谁先击完40下鼠标？

【解析】

根据第1个条件：拉尔夫10秒钟能击10下鼠标；威利20秒钟能击20下鼠标；保罗5秒钟能击5下鼠标，会让人不假思索地认为他们3个人点击鼠标的速度都是每1秒钟点击一次，因此打成平手。

注意，这样的答案是错误的。对于智力类型的题目，不可能有这么简单的推理。如果我们仔细阅读题目，就不难发现还有第 2 个条件，就是他们每人所用的时间的计算方式，即从第一击开始，到最后一击结束。

对于拉尔夫来说，从第一击开始，到最后一击结束一共花了10秒钟，也就是说，他10秒钟能点击9次。同样，威利20秒钟能点击19次，保罗5秒钟能击4次。他们点击鼠标的速度分别为：

拉尔夫0.9次/秒；

威利0.95次/秒；

保罗0.8次/秒；

显然，威利点击鼠标的速度最快，拉尔夫次之，保罗速度最慢。因此，他们不会打成平手，如果比赛点击40下鼠标，当然是威利获胜。

【答案】

他们不会打成平手，威利最先击完40下鼠标。

面试题30 聪明人是怎样发财的

从前有A、B两个相邻的国家，它们的关系很好，不但互相之间贸易交往频繁，而且货币可以通用，汇率也相同。也就是说，A国的100元等于B国的100元。可是两国关系因为一次事件而破裂了，虽然贸易往来仍然继续，但两国国王却互相宣布对方货币的 100 元只能兑换本国货币的90元。有一个聪明人，他手里只有A国的100元钞票，却借机捞了一大把，发了一笔横财。请你想一想，这个聪明人是怎样从中发财的?

【解析】

这道题考查的是应试者能否运用逆向思维来解决问题。

根据题目条件，A、B两个国家由于关系破裂，虽然贸易往来仍然继续，但两国国王却互相宣布对方货币的100元只能兑换本国货币的90元。

首先我们用定向思维来思考，假如我们在A国，要去B国做旅游或者生意，那么拿着A国的钱币去B国，并在B国换成B国的钱币，则只能换带去的钱的9/10。而同样，在B国带着B国的钱去A国交换A国的钱币，也只能换9/10的钱。可见，如果按照这种方式来交换，我们口袋里的钱会越来越少。

现在我们换一个方式来思考：如果我们使用A国的钱在A国内交换B国的钱呢？当然是能换更多的B国的钱。假如我们此时有100元A国的钱币，由于在A国内90元A国钱币能换100元B国钱币，所以在A国内100元A国的钱币能换111元B国的钱币。同样，在B国内100元B国的钱币能换111元A国的钱币。按照这种方式不断在这两个国家进行钱币兑换，就能越换越多。这个聪明人就是用类似这种方式发了一笔横财。

【答案】

在A国内用A国的钱币换取B国的钱币，然后，在B国内把B国的钱币全部换成A国的钱币。如此进行不断交换，钱会越换越多。

面试题31 谁打碎了花瓶

如果下列每个人说的话都是假话，那么是谁打碎了花瓶？

夏克："吉姆打碎了花瓶。"

汤姆："夏克会告诉你谁打碎了花瓶。"

埃普尔："汤姆，夏克和我不太可能打碎花瓶。"

克力斯："我没打碎花瓶。"

艾力克："夏克打碎了花瓶，所以汤姆和埃普尔不太可能打碎花瓶。"

吉姆："我打碎了花瓶，汤姆是无辜的。"

【解析】

由于本题中的六个人说的话都是假话，所以首先可以把他们的话转化为下面的真话。

（1）夏克：吉姆没有打碎花瓶。

（2）汤姆：夏克不会告诉你谁打碎了花瓶。

（3）埃普尔：汤姆，夏克和我都可能打碎花瓶。

（4）克力斯：我打碎了花瓶。

（5）艾力克：夏克没有打碎花瓶，所以汤姆和埃普尔有可能打碎花瓶。

（6）吉姆：我没有打碎花瓶，汤姆打碎了花瓶。

接下来对每个人进行检查。

夏克的检查，在（5）中明确说明了夏克没有打碎花瓶。

汤姆的检查，在（6）中明确说明了汤姆打碎花瓶。

埃普尔的检查，上面6条中都没有明确说明埃普尔是否打碎花瓶，与埃普尔有关的叙述只有（3）和（5），他们都是说埃普尔有打碎花瓶

的可能。但是注意，（3）是埃普尔自己说的话，因此埃普尔打碎了花瓶。

克力斯的检查，在（4）中明确说明了克力斯打碎花瓶。

艾力克的检查，上面 6 条中都没有出现艾力克是否打碎花瓶的记录，因此不能确定艾力克有没有打碎花瓶。

吉姆的检查，在（6）中明确说明了吉姆没有打碎花瓶。

因此得出结论，夏克、吉姆没有打碎花瓶；汤姆、埃普尔、克力斯打碎了花瓶；不能确定艾力克有没有打碎花瓶。

【答案】

夏克、吉姆没有打碎花瓶；

汤姆、埃普尔、克力斯打碎了花瓶；

不能确定艾力克有没有打碎花瓶。

面试题32 大有作为

鲁道夫、菲利普、罗伯特 3 位青年，一个当了歌手，一个考上大学，一个加入美军陆战队，个个未来都大有作为。现已知：

A．罗伯特的年龄比战士的大；

B．大学生的年龄比菲利普小；

C．鲁道夫的年龄和大学生的年龄不一样。

请问：3个人中谁是歌手？谁是大学生？谁是士兵？

【解析】

推理过程如下。

首先可以确定罗伯特是大学生。这是因为由B和C可知，菲利普和鲁道夫的年龄与大学生的年龄都不一样，即他们都不是大学生，所以只有罗伯特是大学生。

当确定了罗伯特是大学生后，就出现了下面两种组合。

（1）鲁道夫是歌手，菲利普是战士。

（2）鲁道夫是战士，菲利普是歌手。

如果第1种组合成立，则原题的3个条件变为：

A．罗伯特的年龄比菲利普大；

B．罗伯特的年龄比菲利普小；

C．鲁道夫的年龄和罗伯特的年龄不一样。

很明显，上面的A和B相矛盾。

最后考虑第2种组合情况，此时原题的3个条件变为：

A．罗伯特的年龄比鲁道夫大；

B．罗伯特的年龄比菲利普小；

C．鲁道夫的年龄和罗伯特的年龄不一样。

此时他们之中，鲁道夫最小，然后是罗伯特，菲利普最大。条件不存在矛盾。因此结论是：罗伯特是大学生，鲁道夫是战士，菲利普是歌手。

【答案】

罗伯特是大学生，鲁道夫是战士，菲利普是歌手。

面试题33 宴会桌旁

在某宾馆的宴会厅里，有4位朋友正围桌而坐，侃侃而谈。他们用了汉、英、法、日4种语言。现已知：

A．甲、乙、丙各会两种语言，丁只会一种语言；

B．有一种语言，4人中有3人都会；

C．甲会日语，丁不会日语，乙不会英语；

D．甲与丙、丙与丁不能直接交谈，乙与丙可以直接交谈；

E．没有人既会日语，又会法语。

请问：甲、乙、丙、丁各会什么语言？

【解析】

推理过程如下。

首先看条件B，有一种语言4人中有3人都会。假设这种语言不是丁会的那种，则甲、乙、丙都会这种语言，可以推出甲、乙、丙 3人能够直接交谈，这与条件D 矛盾（甲与丙不能直接交谈）。因此这种语言一定是丁会的那种语言。再根据条件D，丙与丁不能直接交谈，这说明甲、乙会丁说的语言，而丙不会。

现在根据以上推论的初步结果，可以把上面的A～E条件整理为：

a．甲、乙、丙各会两种语言，丁只会一种语言；

b．甲、乙会丁说的语言，而丙不会丁说的语言；

c．甲会日语，丁不会日语，乙不会英语；

d．甲与丙不能直接交谈，乙与丙可以直接交谈；

e．没有人既会日语，又会法语。

现在推出丁说的是何种语言，步骤如下。

（1）由c可知，丁不会日语，且乙不会英语。这说明丁说的语言是

汉、法之中的一种。

（2）如果丁说的是法语，则由b和c可知，甲会日语和法语，这与e矛盾。

（3）因此丁说的只能是中文。

甲、乙、丙的推断步骤如下。

（1）由于丁说的是汉语，由a、b、c可以很容易推出甲会汉语、日语。

（2）因为甲会汉语、日语，又由d 可知，丙不会汉语、日语，所以丙只会英语、法语（丙会两种语言）。

（3）对于乙，我们知道他会丁说的语言，即汉语，并且他与丙可以直接交谈，这说明他会的另一种是丙会的英、法之中的一种。由条件c 可知，乙不会英语，因此乙会的另一种语言只能是法语。所以乙会汉、法两种语言。

通过上面的推理，我们可以得出下面的结论：

甲会汉、日两种语言；乙会汉、法两种语言；丙会英、法两种语言；丁会汉语。

【答案】

甲会汉、日两种语言；

乙会汉、法两种语言；

丙会英、法两种语言；

丁会汉语。

面试题34 过桥问题

有4个女人要过一座桥。她们都站在桥的某一边，要让她们在17分钟内全部通过这座桥。这时是晚上。她们只有一个手电筒。最多只能让两个人同时过桥。不管是谁过桥，不管是一个人还是两个人，必须要带着手电筒。手电筒必须要传来传去，不能扔过去。每个女人过桥的速度不同，两个人的速度必须以较慢的那个人的速度过桥。

第1个女人：过桥需要1分钟；

第2个女人：过桥需要2分钟；

第3个女人：过桥需要5分钟；

第4个女人：过桥需要10分钟。

比如，如果第1个女人与第4个女人首先过桥，等她们过去时，已经过去了10分钟。如果让第4个女人将手电筒送回去，那么等她到达桥的另一端时，总共用去了20分钟，行动也就失败了。怎样让这4个女人在17分钟内过桥？还有别的什么方法？

【解析】

我们初看此题，很可能会认为，为了让总时间最短，应该始终让第1 个女人进行来回（因为第1个女人回来的时间最短），于是马上会得到下面的方案。

（1）第1个女人和第2个女人过桥，需要2分钟。

（2）第1个女人回来，需要1分钟。

（3）第1个女人和第3个女人过桥，需要5分钟。

（4）第1个女人回来，需要1分钟。

（5）第1个女人和第4个女人过桥，需要10分钟。

一共用时2+1+5+1+10=19分钟。居然比17分钟还多了2分钟。到这

里，许多人会很迷惑，为什么用时不是最短？难道让其他女人做来回快吗？

如果从桥的一边到另一边都用同一个女人来回，那么用第 1 个女人做来回显然是最快的。但是要注意一点，留在另一边的女人也是可以回来的。仔细分析后不难发现，显然让两个最慢的女人（第3个女人和第4个女人）同时过桥更能节省时间。下面为正确的方案：

（1）第1个女人和第2个女人过桥，需要2分钟。

（2）第1个女人回来，需要1分钟。

（3）第3个女人和第4个女人过桥，需要10分钟。

（4）第2个女人回来，需要2分钟。

（5）第1个女人和第2个女人过桥，需要2分钟。

一共用时2+1+10+2+2=17分钟。

面试题35 一句不可信的话

S先生正在家里休息时，接到了一个陌生人打来的预约电话。对方很想在下下个星期的周五去他家里拜访他。但是S先生并不想见这个陌生人，于是他连忙说："下下个礼拜五我非常忙。上午要开会，下午 1 点钟要去参加一个学生的婚礼，接着 4 点钟要去参加一个朋友的孩子的葬礼，随后是我的叔叔的七十寿辰宴会。所以那天我实在是没有时间来接待您的来访了。"

请仔细看题，S先生的话里有一处是不可信的，是哪个地方？

【解析】

由于S先生不想见这个陌生人，因此他说自己下下个礼拜五很忙，并找了以下借口。

（1）上午要开会；

（2）下午1点钟要去参加一个学生的婚礼；

（3）4点钟要去参加一个朋友的孩子的葬礼；

（4）随后是叔叔的七十寿辰宴会。

乍看之下，这4个理由都没有什么漏洞，那么为什么说S先生的话里有一处是不可信的？这一定是因为我们漏掉了什么重要的线索。

如果我们再仔细些，就可以发现本题中还有一个时间限制，也就是陌生人要约定的时间是下下个礼拜五。除此之外没有任何的条件了。

如果现在是礼拜天，下下个礼拜五距离现在有12天，所以陌生人预约的时间距离此刻至少有 12 天的时间。再对上面 4 个借口分析一下，不难得出（3）不可信。这是因为从一个人死后，他的葬礼一般在一个星期内举行，如果定于在至少12天之后举行一个人的葬礼，那么除非大家知道这个人准确的死亡时间。在本题中，S先生借口说12天后要去参

加一个朋友的孩子的葬礼，这显然是不可信的。

【答案】

S先生借口说下下个星期五要去参加一个朋友的孩子的葬礼，这句话不可信。因为谁会知道某个人会一个星期后死亡呢？

面试题36 海盗分宝石

5个海盗抢到了100颗宝石，每一颗都大小一样且价值连城。他们决定这么分：第一步，抽签决定自己的号码（1、2、3、4、5）；第二步，首先，由1号提出分配方案，然后5个人进行表决，当且仅当超过半数的人同意时，按照他的提案进行分配，否则他将被扔入大海喂鲨鱼；第三步，1号死后，再由2号提出分配方案，然后4人进行表决，当且仅当超过半数的人同意时，按照他的提案进行分配，否则他将被扔入大海喂鲨鱼；第四步，以此类推。

条件：每个海盗都是很聪明的人，都能很理智地判断得失，从而做出选择。

问题：最后的分配结果如何？

提示：海盗的判断原则：1．保命；2．尽量多得宝石；3．尽量多杀人。

【解析】

逻辑推理最重要的是要找对思路。

如果从 1 号开始往后（到 5 号结束）推理，则中间的假设非常多，因此很难完成。正确的过程是从后向前推理。

（1）如果1、2、3号强盗都喂了鲨鱼，只剩4号和5号的情况。此时5号一定投反对票让4号喂鲨鱼，这样5号就能独吞所有宝石。

（2）当然，4号也知道当只剩两个人时5号肯定不会支持他，因此为了保命，无论3号给自己多少宝石，他也必须支持3号。

（3）3号也知道4号肯定会支持他，因此4号就会提（100，0，0）的分配方案，即对4号、5号一毛不拔，而将全部宝石归为已有。因为他知道4号一无所获但还是会投赞成票，再加上自己的一票，他的方案即

可通过。

（4）2号推知到3号的方案，就会提出（98，0，1，1）的方案，即放弃3号，而给予4号和5号各一颗宝石。由于该方案对于4号和5号来说，比在3号分配时更为有利，他们将支持他而不希望他出局，从而由3号来分配。这样，2号将拿走98颗宝石。

（5）不过，1号也能推知到2号的方案，并将提出（97，0，1，2，0）或（97，0，1， 0，2）的方案，即放弃2号，而给3号一颗宝石，同时给4号（或5号）2颗宝石。由于1号的这一方案对于3号和4号（或5号）来说，相比2号分配时更优，他们将给1号投赞成票，再加上1号自己的票，1号的方案可获通过，97颗宝石可轻松落入囊中。这无疑是1号能够获取最大收益的方案了！

可以看出，这个推理过程就先考虑简化的极端情况，从而顺藤摸瓜，得出最后的结果。

面试题37 如何推算有几条病狗

村子里有50个人，每人有一条狗。在这50条狗中有病狗（这种病不会传染）。于是人们就要找出病狗。每个人可以观察其他的49条狗，以判断它们是否生病，只有自己的狗不能看。观察后得到的结果不得交流，也不能通知病狗的主人。主人一旦推算出自己家的是病狗，就要枪毙自己的狗，而且每个人只有权利枪毙自己的狗，没有权利打死其他人的狗。第1天，第2天都没有枪响，到了第3天传来一阵枪声。问：有几条病狗，如何推算得出？

【解析】

本题一开始就说50条狗中有病狗，说明了病狗至少1条。

因为可能有多条病狗，为了叙述方便，病狗的主人就用类似病狗主人1、病狗主人2、病狗主人3等来区分。

第1天，病狗的主人1如果看到其他的49个人的家里的狗都没病，而他们50条狗中至少有 1 条，那么就会立即枪毙自己家里的狗，但是第 1 天没有枪声，说明了这个病狗的主人1在其他的49人家里看到至少1条病狗。另外一个有病狗的主人2同样也看到病狗的主人 1 家里有条病狗，所以会认为病狗的主人家里的狗是病狗，而自己家里的不是，所以没开枪。由此，确定病狗数大于1。

第2天，病狗的主人1和病狗的主人2看到昨天没人开枪，如果他们第1天只看到另外49个人家中只有1条病狗，同时他们也确定病狗数大于1，那么他们就会在第2天枪毙自己家里的狗，但是第2天也没有枪声，所以证明他们第1天看到了另外49个人家里还至少有2条病狗，确定病狗数大于2。

第3天，病狗的主人1、2和3看到昨天没人开枪。由文中第3天传来

一阵枪声说明那3个主人已经确定自己家里的是病狗，所以开枪了，这就说明3个病狗的主人在其他人家里各看到2只病狗，所以确定病狗数为3。

【答案】

总共有3条病狗。

面试题38 判断谁是盗窃犯

有个法院开庭审理一起盗窃案件，某地的A、B、C 3 人被押上法庭。负责审理这个案件的法官是这样想的：肯提供真实情况的不可能是盗窃犯；与此相反，真正的盗窃犯为了掩盖罪行，是一定会编造口供的。因此，他得出了这样的结论：说真话的肯定不是盗窃犯，说假话的肯定就是盗窃犯。审判的结果也证明了法官的这个想法是正确的。

审问开始了。

法官先问A："你是怎样进行盗窃的？从实招来！"A回答了法官的问题："叽哩咕噜，叽哩咕噜……"A讲的是某地的方言，法官根本听不懂他讲的是什么意思。法官又问B和C："刚才A是怎样回答我的提问的？叽哩咕噜，叽哩咕噜，是什么意思？"B说："禀告法官，A的意思是说，他不是盗窃犯。"C说："禀告法官，A刚才已经招供了，他承认自己就是盗窃犯。"B和C说的话，法官是能听懂的。听了B和C的话之后，这位法官马上断定：B无罪，C是盗窃犯。

请问：这位聪明的法官为什么能根据 B 和 C 的回答，做出这样的判断？A 是不是盗窃犯？

【解析】

根据本题条件，说真话的肯定不是盗窃犯，说假话的肯定就是盗窃犯，A、B、C 3 人有一个人是盗窃犯，因此只有他一个人会说假话，其他两个人都会说真话。

推理过程如下。

（1）首先是A的判断。如果A不是盗窃犯，那么A是说真话的，这样他会说自己"不是盗窃犯"；如果A是盗窃犯，那么A是说假话的，这样他也必然会说自己"不是盗窃犯"。因此，A 对于法官的提问，他的回

答肯定都是否定的，即说自己不是盗窃犯。因此这里不能判断A是否为盗窃犯。

（2）然后是B的判断。B禀告法官说A不是盗窃犯。由（1）可知，不管A是不是盗窃犯，他的回答都是否定的，由此可知B如实地禀告了法官，所以B说的是真话，因此B不是盗窃犯。

（3）最后是C的判断。C说A承认自己是盗窃犯，即说的与B不同。由于B说的是真话，因此C说的必然是假话，所以C肯定是盗窃犯。

【答案】

法官问A的问题，不管A是否为盗窃犯，A的回答都一定是否定的。在这种情况下， B如实地转述了A的话，而C没有。因此B说了真话，C说了假话。不能判断A是否为盗窃犯。

面试题39 向导

在大西洋的“说谎岛”上，住着X、Y两个部落。X部落总是说真话，Y部落总是说假话。

有一天，一个旅游者来到这里迷路了。这时，恰巧遇见一个土著人A。

旅游者问：“你是哪个部落的人？”

A回答说：“我是X部落的人。”

旅游者相信了A的回答，就请他做向导。

他们在途中看到远处的另一位土著人B，旅游者请A去问B是属于哪一个部落的。A回来说：“他说他是X部落的人。”旅游者糊涂了。他问同行的逻辑博士：“A是X部落的人，还是Y部落的人呢？”逻辑博士说：“A是X部落的人”。

为什么？

【解析】

我们对A是X部落的人和A不是X部落的人这两种情况分别进行分析。

假设A是X部落的人。

（1）如果A遇见的B是X部落的人，由于X部落的人是说真话的，那么，B就说自己是X部落的人，这时，A向旅游者如实地传达了这个回答。

（2）如果A遇见的B是Y部落的人，由于X部落的人是说假话的，那么，B也会说自己是X部落的人，这时，A也向旅游者如实地传达了这个回答。

假设A是Y部落的人。

（1）如果A遇见的B是X部落的人，由于X部落的人是说真话的，那么，B就说自己是X部落的人。由于A是Y部落的人，他是说假话的，所以，他会把B的回答向旅游者传达为“B说他是Y部落的人”。

（2）如果A遇见的B是Y部落的人，由于Y部落的人是说假话的，那么，B就说自己是X部落的人，而A是Y部落的人，也会把B的回答传达为“他说他是Y部落的人”。

从题目的给定条件可知，A 对旅游者传达的话是 “他（指 B）说他是 X 部落的人”。可见，假定A是Y部落的人时得出的（1）、（2）两个结论，都是与题目给定的条件相矛盾的；只有前一个假定（假定A是X部落的人），才符合题目给定的条件。所以，做向导的A是X部落的人。

【答案】

不管B是X部落的人还是Y部落的人，B都会说他自己是X部落的人，A如实地转述了B的话，因此A是X部落的人。

面试题40 扑克牌问题

S先生、P先生、Q先生都具有足够的推理能力。这天，他们正在接受推理面试。

他们知道桌子的抽屉里有如下16张扑克牌：

红桃A、Q、4

黑桃J、8、4、2、7、3

梅花K、Q、5、4、6

方片A、5

约翰教授从这16张牌中挑出一张牌来，并把这张牌的点数告诉P先生，把这张牌的花色告诉Q先生。这时，约翰教授问P先生和Q先生："你们能从已知的点数或花色中推知这张牌是什么牌吗？"

于是，S先生听到如下对话。

P先生："我不知道这张牌。"

Q先生："我知道你不知道这张牌。"

P先生："现在我知道这张牌了。"

Q先生："我也知道了。"

听罢以上的对话，S先生想了一想之后，就正确地推断出这张牌是什么牌。请问：这张牌是什么牌？

【解析】

P先生知道这张牌的点数，Q先生知道这张牌的花色。推理过程如下。

（1）首先由P先生说"我不知道这张牌"以及随后Q先生对P先生说"我知道你不知道这张牌"可知，这张牌的点数与其他牌的点数有重复。由于点数 J、8、2、3、7、K、6都是唯一的，因此可以过滤掉，于

是筛选出只有点数重复的牌，如下所示。

红桃A、Q、4

黑桃4

梅花Q、5、4

方片A、5

（2）然后P先生和Q先生都说知道什么牌了。在（1）中筛选后的牌中，A、Q、5在红桃、梅花、方片中各有一张并且这3种花色至少有两种点数，因此点数如果是A、Q、5中的一个，P 先生和 Q 先生无论如何都是猜不出来的。在这里，只有花色为黑桃的点数只有一种，因此这张牌是黑桃4。

【答案】

这张牌是黑桃4。

面试题41 谁是机械师

一列火车上有3个工人，即史密斯、琼斯、罗伯特，3人的工作为消防员、司闸员、机械师，有3个乘客与3人名字相同。

（1）罗伯特住在底特律。

（2）司闸员住在芝加哥和底特律中间的地方。

（3）琼斯一年赚2万美金。

（4）有一个乘客和司闸员住在一个地方，每年的薪水是司闸员的3倍整。

（5）史密斯台球打得比消防员好。

（6）和司闸员同名的乘客住在芝加哥。

请问：谁是机械师？

【解析】

这里一共有6个人，他们的名字为史密斯、琼斯、罗伯特。下面是推理过程。

首先要搞清楚和司闸员住一个地方的乘客的名字。

（1）条件（3）中，因为2万美金不能被3整除，这说明那个乘客不是琼斯（当然还有第2个琼斯）。于是，他的名字可能为史密斯或者罗伯特。

（2）条件（1）中，罗伯特住在底特律。这里注意题目中并没有指明是哪个罗伯特（是工人还是乘客），因此这就是说两个罗伯特都住在底特律。所以和司闸员住一个地方的乘客名字必定是史密斯。

接下来能推出司闸员的名字。条件（6）中，和司闸员同名的乘客住在芝加哥，罗伯特住在底特律，所以司闸员既不叫史密斯又不叫罗伯特，司闸员就是琼斯。

最后推出机械师的名字。条件（5）中，史密斯不是消防员，并且他也不是司闻员，因此史密斯就是机械师。

【答案】

史密斯是机械师。

面试题42 帽子的颜色

10个人排队戴帽子，10个黄帽子，9个蓝帽子，戴好后，后面的人可以看见前面所有人的帽子，然后从后面问起，问自己头上的帽子是什么颜色，结果一直问了9个人，都说不知道，而最前面的人却知道自己头上的帽子的颜色。问：他们的帽子分别是什么颜色，为什么？

【解析】

最前面的那个肯定戴着黄帽子。因为后面有九个人都不能确定自己是什么帽子，说明有可能是黄帽子，有可能是蓝帽子。

本题要倒推才行。

假如队中只有2个人，此时有2个人戴黄帽子，1个人戴蓝帽子。

如果第 1 个人戴蓝帽子，因为第 2 个人可以看见前面的，而蓝帽子只有一个。这种情况第2个人可以推断出自己戴黄帽子。如果第1个人戴黄帽子，则第2人戴的是黄帽子还是蓝帽子就不确定。

假如队中有3个人，此时有3个人戴黄帽子，2个人戴蓝帽子。在这种情况下有以下几种可能。

（1）1蓝、2蓝的情况下，3知道自己的帽子是什么颜色，因为只有两个蓝帽子。

（2）1 蓝、2 黄的情况下，3 不知道自己的帽子是什么颜色，2 知道自己的帽子是什么颜色。因为2会这样思考，1的是蓝色，如果自己的帽子是蓝色的话，那么3应该知道自己的帽子是什么颜色，而3不知道，则自己的帽子肯定是黄色。

（3）1 黄、2 蓝的情况下，3 不知道自己的帽子是什么颜色，2 也不能确定自己的帽子是什么颜色。

（4）1黄、2黄的情况下，2和3都不确定自己的帽子是什么颜色。

排除A和B的情形，只剩C和D。在这两种情况下，1的帽子都是黄色。

以此类推……

所以在10个人的情况下，只有第1个人戴黄帽子，其他人的不能确定。如果第1个人戴蓝帽子，则剩下的9个人中，必定有一个人能确定。

【答案】

在10个人的情况下，只有第1个人戴黄帽子，其他人的不能确定。如果第1个人戴蓝帽子，则剩下的9个人中，必定有一个人能确定。

面试题43 两个大于1小于10的整数

两个大于1小于10的整数，把两数之和告诉甲，两数之积告诉乙。让他俩猜，两人都说不知道。之后两人都沉思了一会儿。突然乙说“我知道这两个数了”，甲也跟着说“我知道了”。请问：这两个数各是多少？

【解析】

由题意可知，这两个整数在2～9之中。

把2～9的数分别与2～9的数相加，结果如下。

加2：4、5、6、7、8、9、10、11

加3：5、6、7、8、9、10、11、12

加4：6、7、8、9、10、11、12、13

加5：7、8、9、10、11、12、13、14

加6：8、9、10、11、12、13、14、15

加7：9、10、11、12、13、14、15、16

加8：10、11、12、13、14、15、16、17

加9：11、12、13、14、15、16、17、18

把2～9的数分别与2～9的数相乘，结果如下。

乘2：4、6、8、10、12、14、16、18

乘3：6、9、12、15、18、21、24、27

乘4：8、12、16、20、24、28、32、36

乘5：10、15、20、25、30、35、40、45

乘6：12、18、24、30、36、42、48、54

乘7：14、21、28、35、42、49、56、63

乘8：16、24、32、40、48、56、64、72

乘9：18、27、36、45、54、63、72、81

甲知道两数之和，乙知道两数之积。

很明显，甲在乙之后确定这两个数，也就是说，甲刚开始并不能确定。这说明两数之和在上面的相加列表中不唯一，可以推出两数之和应该在6～16之中。因为4只能是2+2， 5 只能是 2+3，17 只能是 8+9，18 只能是 9+9，而其他的和数有至少两种相加组合，比如7=3+4或7=2+5，16=8+8或16=7+9。

乙首先说自己知道这两个数了，这说明两数之积在上面相乘的列表中是唯一的（除去对称性相同，比如3×5=5×3）。因此可以得到以下积数。

4、6、8、10、14、15、21、25、27、30、35、

40、42、45、48、54、56、63、64、72、81。

而甲随后也跟着说他也知道了。也就是说，当他知道两数之积在上面相乘的列表中唯一的时候，他就能确定两个数。由于两数之和应该在6～16 之中，因此上面的积数可以首先除去4、6、72、81。剩下的积数为：

8、10、14、15、21、25、27、30、35、

40、42、45、48、54、56、63、64。

现在对每一个积数进行分解：

（1）8：8=2×4，2+4=6

（2）10：10=2×5，2+5=7

（3）14：14=2×7，2+7=9

（4）15：15=3×5，3+5=8

（5）21：21=3×7，3+7=10

（6）25：25=5×5，5+5=10

（7）27：27=3×9，3+9=12

（8）30：30=5×6，5+6=11

（9）35：35=5×7，5+7=12

（10）40：40=5×8，5+8=13

（11）42：42=6×7，6+7=13

（12）45：45=5×9，5+9=14

（13）48：48=6×8，6+8=14

（14）54：54=6×9，6+9=15

（15）56：56=7×8，7+8=15

（16）63：63=7×9，7+9=16

（17）64：64=8×8，8+8=16

显然，只有上面（1）～（4）中的和数没有重复，即这两个数有以下几种组合：2和4、2和5、3和5、3和7。

【答案】

这两个数可能为：

2和4、

2和5、

3和5

或3和7。

面试题44 谁用1美元的纸币付了糖果钱

美国货币中的硬币有1美分、5美分、10美分、25美分、50美分和1美元这几种面值。请看正文，挑战你逻辑推理的极限：

一家小店刚开始营业，店中只有 3 位男顾客和一位女店主。当这 3 位男士同时站起来付账的时候，出现了以下情况。

（1）这4个人每人都至少有一枚硬币，但都不是面值为1美分或1美元的硬币。

（2）这4人中没有一人能够兑开任何一枚硬币。

（3）一个叫卢的男士要付的账单款额最大，一位叫莫的男士要付的账单款额其次，一个叫内德的男士要付的账单款额最小。

（4）每个男士无论怎样用手中所持的硬币付账，女店主都无法找清零钱。

（5）如果这 3 位男士相互之间等值调换一下手中的硬币，则每个人都可以付清自己的账单且无须找零。

（6）当这 3 位男士进行了两次等值调换以后，他们发现手中的硬币与各人自己原先所持的硬币没有一枚面值相同。

随着事情的进一步发展，又出现如下情况。

（7）在付清了账单而且有两位男士离开以后，留下的男士又买了一些糖果。这位男士本来可以用他手中剩下的硬币付款，可是女店主却无法用她现在所持的硬币找清零钱。

（8）于是，这位男士用 1 美元的纸币付了糖果钱，但是现在女店主不得不把她的全部硬币都找给了他。

现在，请你不要管那天女店主怎么会在找零上屡屡遇到麻烦，思考：这 3 位男士中谁用1美元的纸币付了糖果钱？

【解析】

推理过程如下。

首先根据条件（6），必定先有一位男士（称之为男士A）和另一位男士（称之为男士B）调换了硬币，然后男士B必定和第3位男士（称之为男士C）调换了硬币；在这些调换中，男士A必定把他的全部硬币都换给了男士B。因此，男士A手中所持的全部硬币一定可以用硬币的两种组合来表示。

根据条件（1），每种组合中都不包括1美分和1美元的硬币。

根据条件（2），每种组合中的硬币都不能兑开一枚较大面值的硬币。

根据条件（6），这两种组合之间没有一枚硬币面值相同。

经过对满足这3条要求的硬币组合的寻找，可以发现男士A开始和最后持有的硬币只可能有两种总额：

一种总额是55 美分。它有下面两种组合：（A）一枚25 美分硬币和三枚10 美分硬币；（B）一枚50美分硬币和一枚5美分硬币。

另一种总额是30 美分。它有下面两种组合：（A）三枚10 美分硬币，（B）一枚25美分硬币和一枚5美分硬币。

因此，如果N代表5美分硬币，D代表10美分硬币，Q代表25美分硬币，H代表50美分硬币，则男士A开始时持有的全部硬币和男士B开始时持有的部分或全部硬币必定是下列四种情况之一。

（I）QDDD、HN；

（II）HN、QDDD；

（III）DDD、QN；

（IV）QN、DDD。

根据条件（6），在随后男士C和男士B调换时，他一定把手中所持的全部硬币都换给了男士B。如果男士C持有上面四种硬币组合中的任何一种组合，那么男士B将从男士C手中换来与他换给男士A的某些硬币

面值相同的硬币，从而与（6）矛盾。所以男士C持有的组合不是上面四种硬币组合中的任何一种组合。

因此，男士C从男士B手中换来的硬币必定能兑开他开始就有的某枚硬币。这样，男士B换给男士C至少一枚他从男士A手中换来的硬币和至少一枚他自己开始就有的硬币。不然的话，男士A或男士B一开始就能兑开某种面值的一枚硬币，从而与条件（2）矛盾。所以，至少有一枚硬币过了3个人的手。这是一枚什么样面值的硬币呢？

由于没有一个人有1 美元的硬币，所以过3 个人的手的硬币不会是50 美分的。

如果过3 个人的手的是一枚5 美分的硬币，则（II）或（IV）代表男士A 和男士B之间的调换。可是这样一来，男士B要兑换一枚较大面值的硬币，手中还得有两枚10美分的硬币或一枚5美分的硬币，从而与条件（2）矛盾。因此，过3个人的手的不是5美分的硬币。

如果过3 个人的手的是10 美分的硬币，则（I）或（III）代表男士A和男士B 之间的调换。可是这样一来，男士 B 要兑开一枚较大面值的硬币，手中还得有两枚 10美分的硬币或一枚5美分的硬币，从而与条件（2）矛盾。因此，过3个人的手的不是10美分硬币。

所以，过3 个人的手的只可能是25 美分的硬币。

也就是说，（I）或（IV）代表了男士A和男士B之间的调换。在这两种情况下，为了不与条件（2）矛盾，在情况（I）下，男士B不能再有两枚10美分的硬币，或在情况（IV）下一枚5美分的硬币。男士B也不能再有一枚10美分的硬币，因为他无法用这枚10美分的硬币。加上他从男士A那儿换来的任何硬币去调换一枚较大面值的硬币，从而与条件（6）矛盾。在情况（I）下，他也不能再有一枚50美分的硬币，因为那会与（2）矛盾。

所以现在3位男士所持有的硬币只有3种可能的情况：

（I）QDDD、HN、Q;

（IVa）QN、DDD、Q；

（IVb）QN、DDD、HQ。

由条件（1）可知，由于过3个人的手的25美分硬币不会用于调换1美元的硬币，所以这枚25美分的硬币必定用于调换50美分的硬币。因此，在男士A和男士B调换之后，男士C一定是给了男士B一枚50美分的硬币，换来了至少一枚25美分的硬币。但在情况（I）和（IVb）下，这样的调换结果与条件（6）矛盾。于是，只有（IVa）是符合实际的持币情况。

根据条件（4）和（5），男士A的账单数额必定是10美分或20美分，男士B的账单数额必定是5美分或50美分，男士C的账单数额必定是25美分。于是，符合实际情况的账单必定是下列四组账单之一。

（i）20美分、5美分、25美分；

（ii）10美分、5美分、25美分；

（iii）10美分、50美分、25美分；

（iv）20美分、50美分、25美分。

首先（i）是不可能的，因为根据条件（1），女店主开始时至少有一枚硬币（非1美分）；根据条件（8），在3份账单付清后，她的硬币总额小于1美元。如果（i）符合实际情况，则在收清3份账单之前，女店主没有25美分的硬币。而且，她也没有5美分的硬币（根据男士B 开始持有的硬币和条件（4）），也没有50 美分的硬币（根据条件（8）），也没有10美分的硬币（根据男士 A 开始时持有的硬币和条件（4））。因此（i）与（1）（她至少有一枚硬币）矛盾，从而（i）是不可能的。

如果（ii）符合实际情况，则女店主没有5美分的硬币（根据男士B开始时持有的硬币和条件（4）），也没有25美分的硬币（根据男士C开始时持有的硬币和条件（4）），也没有50美分的硬币（根据男士B开始时持有的硬币和条件（4）），也没有两枚或多于两枚的10美分的硬币

（根据男士A开始时持有的硬币和条件（4））。因此在这种情况下，她应该只有一枚10美分的硬币。

如果（iv）符合实际情况，则在3份账单付清后，女店主有硬币QDDN，男士A有DD，男士B有H，男士C有Q。根据条件（8），女店主所有硬币的总额与1美元之差等于糖果的价钱。因此糖果的价钱是50美分。但是，根据条件（7），买糖果的那位男士所有的硬币总额超过糖果的价钱。这样，没有一位男士买了糖果（因为这时每位男士的硬币总额都没有超过50美分）。于是，（ii）是不可能的。

现在只有（iii）才可能符合实际的情况。根据条件（3），内德是男士A，卢是男士B，莫是男士C。在付清3份账单后，女店主有硬币HQDD，内德有硬币DD，卢有硬币N，莫有硬币 Q。根据条件（8），糖果的价钱一定是 5 美分。于是，根据条件（7），买糖果的既不是有5美分的卢，也不是有25美分的莫。这样，是有两枚10美分硬币的内德买了糖果。因此，“是内德给了女店主1美元的纸币。”

全部的情况，可以总结如下。

开始时持有：卢QDDD，莫H，内德QN；

第1次调换后持有：卢QQN，莫H，内德DDD；

第2次调换后持有：卢HN，莫QQ，内德DDD；

付账后持有：卢HN，莫Q，内德DD；

糖果的价钱：5美分。

【答案】

开始时持有：卢QDDD，莫H，内德QN；

第1次调换后持有：卢QQN，莫H，内德DDD；

第2次调换后持有：卢HN，莫QQ，内德DDD；

付账后持有：卢HN，莫Q，内德DD；

糖果的价钱：5美分。

给女店主1美元的纸币的人是内德。

面试题45 究竟有哪些人参加了会议

有人邀请A、B、C、D、E、F六个人参加一项会议，这6个人有些奇怪，因为他们有很多要求：

（1）A、B至少有一个人参加会议；

（2）A、E、F 3 人中有两个参加会议；

（3）B和C两个人一致决定，要么两人都参加，要么两人都不参加；

（4）A、D两人中只有一人参加；

（5）C、D两人中也只有一人参加；

（6）如果D不去，那么E也决定不去。

那么最后究竟有哪些人参加了会议呢？为什么？

【解析】

推理步骤如下。

（1）首先判断D会不会去参加会议。

假如D参加会议，则可进行下面推断。

根据条件（4）和（5），可知A、C都不去参加会议。

再由条件（3），可知由于C不去导致B也不去。

因此A、C、B都不会去参加会议。这与条件（1）矛盾（A、B至少有一个人参加会议）。

至此推断D一定没有参加会议。

（2）D没有参加，根据条件（4）推出A必然参加。

（3）D没有参加，根据条件（5）推出C也必然参加。

（4）D没有参加，根据条件（6）推出E必然没有参加。

（5）由于C参加，根据条件（3）推出B必然参加。至此推出A、

B、C参加会议，而D、E没有参加会议。只有F没有进行判断。

（6）由于A参加，而E没有参加，所以根据条件2可知，F必然参加会议。

【答案】

参加会议的有A、B、C、F。D和E没有参加会议。

面试题46 小虫

有一种小虫，每隔两秒钟分裂一次。分裂后的两只新的小虫经过两秒钟后又会分裂。如果最初某瓶中只有一只小虫，那么两秒后变两只，再过两秒后就变 4 只……两分钟后，正好满满一瓶小虫。现在这个瓶内最初放入两只这样的小虫。

问：经过多长时间后，正巧也是满满的一瓶？

【解析】

为了进行快速答题，这里我们不需要考虑总共需要小虫在规定时间内裂变多少次，或者这个瓶子总共能装多少只小虫。

实际上，我们只需要考虑两个条件：

裂变的规律是两秒钟1 只小虫变成两只小虫。

原来是一只小虫需要2 分钟裂变装满小瓶，现在是两只小虫裂变。

因此，小虫是以2的乘方的速度增长的，而现在与原来的区别是初始为2，就是总裂变比原来少了一次，也就是比原来缩短了2秒钟，即1分58秒。

【答案】

1分58秒。

面试题47 相遇

美国某小镇车队有17辆小公共汽车，整天在相距197千米的青山与绿水两个小镇之间往返运客。每辆车到达小镇后司机都要休息8分钟。司机杰克上午10时20分开车从青山镇出发，在途中不时地遇到（有时是迎面驶来，有时是互相超越）一辆本车队的车。下午1时55 分他到达绿水镇，休息时发现本队的其他司机一个都不在。没有同伴可以聊天，杰克就静静地回忆刚才在路上遇到的本车队的那些人。

问：杰克一共遇到了本车队的几辆车？

【解析】

本题中有很多条件，但绝大多数都与解体无关。比如：

青山与绿水两个小镇之间相距197 千米。

每辆车到达小镇后司机都要休息8 分钟。

司机杰克上午10 时20 分开车从青山镇出发。

下午1 时55 分他到达绿水镇。

以上这些条件看似告诉了一些有用的数据，但实际上有用的条件只有下面两个。

小镇车队共有17 辆小公共汽车。

休息时发现本队的其他司机一个都不在。

由于司机杰克的车也属于这种小公共汽车，因此他遇到的是其他司机开的车，即总共为17-1=16辆车。

【答案】

杰克在路上遇到了其他车的司机，所以一共16辆小公共汽车。

面试题48 约会

矩阵博士的女儿艾娃小姐是他和日本夫人的独生女，她真是位绝佳美人。怪不得马丁先生对她动心了。不过，这位小姐生性羞怯，如果直截了当地请她吃饭，可能会遭到谢绝。对此，马丁先生绞尽了脑汁，苦思对策。

突然间，他心血来潮，想起了哈佛大学的数学家吉尔比贝克教给他的锦囊妙计，顿时心花怒放，喜上眉梢。

“亲爱的，我有两个问题要问您，而且都只能回答：‘是’或‘不’，不准用其他语句。但在正式提问以前，我要同您预先讲好，您一定要听清楚之后再郑重回答，而且两个问题的答案都必须在逻辑上是完全合理的，不能自相矛盾。”他对艾娃说。

艾娃略微蹙了一下眉，感到非常有趣，于是，她爽朗地说：“好吧！那就请您发问吧！”

问：马丁先生该怎样提问，才能达到请艾娃小姐吃饭的目的？

【解析】

为了达到请艾娃小姐吃饭的目的，马丁先生必须用艾娃小姐对于第1 个问题的回答来限制第2个问题的答案。所以他的第1个问题必须含有与第2个问题相关的内容。比如：

两个问题你都回答“不”吗？

两个问题你都回答“是”吗？

我的第2 个问题你要回答“不”吗？

我的第2 个问题你要回答“是”吗？

然后马丁先生就可以针对艾娃小姐的问答设计第 2 个问题。比如艾娃小姐对于“两个问题你都回答“不”吗？”的回答是“是”，那么马丁先生

的第2个问题就是“你不答应同我吃饭吗？”，如果艾娃小姐的回答是“不”，那么马丁先生的第 2 个问题就是“你答应同我吃饭吗？”这样无论艾娃小姐如何回答，都只能答应马丁先生的请求了。

【答案】

第1个问题是：两个问题你都回答“不”吗？

如果艾娃小姐的回答是“是”，第2个问题是：你不答应同我吃饭吗？

如果艾娃小姐的回答是“不”，第2个问题是：你答应同我吃饭吗？

面试题49 30秒答题

（1）你在什么地方总能找到幸福？
（2）一个人走进他的花园时，总是把什么先放在里边？
（3）什么东西越洗越脏？
（4）什么东西能载得动一百捆干草却托不起一粒沙子？
（5）什么东西越是打破了越是受人欢迎？
（6）在早餐时从来不吃的是什么？
（7）放大镜不能放大的东西是什么？
（8）什么东西倒立后会增加一半？
【答案】
（1）字典里。
（2）脚。
（3）水。
（4）水。
（5）纪录。
（6）晚餐。
（7）角度。
（8）数字6。

面试题50 1分钟答题

（1）当您从西向东行走时，不久向左转 270 度角行走，再向后转走，接着，又向左转90度角走，最后又向后转走。请问：最终您是朝哪一个方向行走的？

（2）在20世纪有这样一个年份，把它写成阿拉伯数字时，正看是这一年，倒过来看还是这一年。请问：这是指哪一个年份？

（3）用三根火柴要摆成一个最小的数（不许把火柴折断或弯曲），这个数是多少？

（4）有一个又高又狭窄的玻璃筒，筒里放着一只鲜鸡蛋。如果不许把玻璃筒倾斜，也不许用任何夹具把鲜鸡蛋夹起，那么，您有什么办法取出鲜鸡蛋？

（5）英国伦敦某公司采购员杰夫经常出差去法国巴黎，而且每次都是乘坐火车去的。有一次，他又要出差去法国巴黎，但他前一半路程是坐飞机去的，这比他平常坐火车去的速度要快八倍；而他后一半路程是坐火车和汽车到达法国巴黎的，速度比他平常坐火车要慢一半。请问：他这一次出差去法国巴黎，是否比他平常坐火车去节省时间？为什么？

（6）一只走着的挂钟，它在24小时里，分针和时针要重合多少次？

（7）如果给您一根较长的粗铜线，要用这根铜线将燃着的蜡烛火焰熄灭，但又不许您用铜线碰到蜡烛，请问：有何办法？

（8）有一根铁线，如果用钳子把它剪断后，它仍然是一根与原来长度相等的铁线。请问：这是一根什么形状的铁线？

（9）宇航员卡特在乘宇宙飞船进入太空前，正用他所带的自来水

笔为来访者签名留念。当他进入太空以后，他正忙着用这支笔写日记。您相信吗？

（10）有12个人要过河去，河边只有一条能够载3个人的小船。请问：这12个人都过河，需要渡几次？

【解析】

题（1），向左转270度角后，方向是向南；再向后转走，方向是向北；又向左转90度角走，方向是向西；最后又向后转走，方向向东。因此最后朝东行走。

题（2），数字0～9中只有数字1和数字8是上下对称的，因此年份为1881。

题（3），负数小于0，因此三根火柴有一根用作符号，显然剩余两根摆成11，即-11。

题（4），显然我们不可以直接取出鲜鸡蛋，所以只有让鸡蛋自己出来，可以增加水的浮力（比如加醋）。

题（5），本题中不需要计算坐飞机比坐火车节省了多长时间，只需要注意后一半路程他是坐火车和汽车到达法国巴黎的，速度比他平常坐火车要慢一半，也就是说，后一半路程花的时间等于他平常坐火车去的总时间。再加上飞机的时间，显然比平常坐火车去多花了时间。

题（6），0时起碰到的时间大致为：

0:00、1:05、2:11、3:16、4:22、5:27、6:33、7:38、8:43、9:49、10:54，到这里有11次。

然后又从头来11次，一共22次。

如果最后到达0：00也计算在内，一共就是23次。

题（7），可以把铜线绕成一个扇子形状，然后扇风将蜡烛吹灭。

题（8），这个铁线是环状的，剪断后铁线没有了环，但长度和原来相同。

题（9），进入太空之后，由于没有了地球的引力，所有东西都失

去了重量，自来水笔中的墨水也是如此。因此不可能用自来水笔写日记。

题（10），这里相当于一个船夫运 11 个人，去河对岸可坐 3 个人，从河对岸回来至少需要一个人（作为船夫）。所以每次送两个人去河对岸，总共需要渡6次。

面试题51 现代斯芬克斯之谜

斯芬克斯是古代希腊神话中的带翅膀的狮子女魔。传说她在底比斯附近要人猜谜，猜不出来就要杀人。一次，她邀请底比斯王子猜谜："有一种动物，早上四条腿，中午两条腿，晚上三条腿，是什么动物？"聪明的王子说："是人。"他猜中了。

如果你是现代的斯芬克斯，会提出什么样的问题呢？比如，1和0之间加上什么符号才可以使得到的数比0大又比1小呢？你知道吗？

【答案】

人的一生：生下来只能爬，所以有四条腿；中年时成年了，当然两条腿；晚年时老了，要拐杖，所以三条腿。

1和0之间加上小数点"."后变成0.1，0.1比0大又比1小。

面试题52 所有开着的灯的编号

有100盏灯，从1～100编上号，开始时所有的灯都是关着的。第1次，把所有编号是1的倍数的灯的开关状态改变一次；第2次，把所有编号是2的倍数的灯的开关状态改变一次；第 3 次，把所有编号是 3 的倍数的灯的开关状态改变一次；依此类推，直到把所有编号是100的倍数的灯的开关状态改变一次。问：此时所有开着的灯的编号有哪些？

【解析】

本题中灯的数量是100盏，因此我们不可能进行100次试验后得到结果。

显然，100 盏灯是否开着与它们状态改变的次数的奇偶性有关。由于初始时所有的灯都是关着的，因此如果改变次数为奇数，则灯是开着的；如果改变次数为偶数，则灯是关着的。

如何判断各个小灯的状态改变的次数是奇数还是偶数呢？我们知道任何一个数可以分解成 1 和它自身，也有可能分解成另外两个较小的数的乘积，由于是两两相乘，因此可知一般都是偶数次数。只有一种情况例外，就是这个数能表示为某个数的2次方。比如，9可以分解成3的2次方，即3×3。

显然，1～100的数只可能出现1～10的乘方。因此所有开着的灯的编号为

1、4、9、16、25、36、49、64、81、100。

【答案】

所有开着的灯的编号为1、4、9、16、25、36、49、64、81、100。

图书在版编目（**CIP**）数据

C和C++程序员面试秘笈/董山海编著.--北京：人民邮电出版社，2014.3

ISBN　978-7-115-34113-6

Ⅰ.①C…　Ⅱ.①董…　Ⅲ.①C语言—程序设计　Ⅳ.①TP312

中国版本图书馆CIP数据核字（2013）第307367号

内容提要

众多高级语言都从 C/C++有所借鉴，所以说 C/C++的语言基础对从事软件开发的人员来说非常重要。

本书是一本解析 C/C++面试题的书，可以帮助求职者更好地准备面试。本书共包含12 章，囊括了目前企业中常见的面试题类型和考点，包括 C/C++程序基础，预处理、const、static 与 sizeof，引用和指针，字符串，位运算与嵌入式编程，C++面向对象，C++继承和多态，数据结构，排序，泛型编程，STL，算法和逻辑思维等最常见的面试题。本书通过技术点解析、代码辅佐的方式让读者能深刻领会每个考点背后的技术。

本书紧扣面试精髓，对各种技术的剖析一针见血，是目前想找工作的 C/C++程序员和刚毕业的大学生的面试宝典。

◆编著　董山海

责任编辑　陈冀康

责任印制　程彦红　焦志炜

◆人民邮电出版社出版发行　　北京市丰台区成寿寺路11号

邮编　100164　　电子邮件　315@ptpress.com.cn

网址　http://www.ptpress.com.cn

北京昌平百善印刷厂印刷

◆开本：800×1000　1/16

印张：29

字数：564千字　　2014年3月第1版
印数：1–3000册　　2014年3月北京第1次印刷
定价：59.00元
读者服务热线：**(010)81055410**　印装质量热线：**(010)81055316**
反盗版热线：**(010)81055315**